Project in Data Compression Techniques (Spring 2012)

Topics to be shared Tue 13.3, 14-16, B119, Veli Mäkinen and Simon Puglisi

1. **Adaptive Huffman Coding**

   Implement fully working adaptive huffman coding algorithm as described at the lectures (with constant time updates to the tree).

2. **Arithmetic Coding**

   Implement fully working arithmetic coding algorithm as described at the lectures (that scales to long texts).

3. **Relative Lempel-Ziv and Backward Searching**

   Relative Lempel-Ziv factorization is the same as normal Lempel-Ziv but the references always go to a given fixed string. For example, let $A =$ `abcba` and $B =$ `abba`. Then $Z(B \mid A) = (1,2)(4,2)$, where x in (x,y) denotes position in A and y in (x,y) denotes the length. That is, $(1,2)$ corresponds to the underlined substring $\underline{\texttt{ab}}\texttt{cba}$ and $(4,2)$ to the underlined substring $\texttt{abc}\underline{\texttt{ba}}$ in our example. The factorization is easy to produce in linear time by building first the suffix tree of A. Let $B[1 \ldots i]$ be already factorized. Then one can find the maximal prefix of $B[i+1 \ldots]$, say $B[i+1 \ldots j]$, such that there is a path in suffix tree of A matching it. Printing any suffix number from the found subtree and the length $j - i$ gives the next factor. Then the process can be repeated setting $i = j + 1$.

   It turns out that suffix tree can be replaced by a more space-efficient data structure in this construction. Build an FM-index for the reverse of A. Then doing backward search on the reverse of B analogously to the suffix tree algorithm above gives the same factorization: a path in suffix tree corresponds to a non-empty suffix array interval found by the backward search.

   The goal is to implement relative Lempel-Ziv coding using an existing implementation of FM-index (for example, taken from Pizza-Chili corpus). The emphasis is on finding suitable integer codes for the Lempel-Ziv tuples on data sets where B is very similar to A.

4. **Succinct Graph Representations and Graph Algorithms**

   Implement the mechanism given at lectures to represent the adjacency matrix of a graph through bitvectors, supporting navigation in the graph. Use suitable (gap-encoded) bitvector library implementation to guarantee succinct representation on sparse graphs. Implement some simple graph algorithms (e.g. Euler tour, lightest path in acyclic weighted directed graph) on top of the graph representation. Pay attention to the size of the external data structures required by the chosen graph algorithms.

5. **Dictionary Compression and Generation of Random Texts**

   Implement the static dictionary example given at lectures, where a dictionary of english words is given with their typical frequences, and the goal is to encode a new text given the static dictionary. Notice that this gives a nice way of generating (somewhat) correct looking random english texts; generate a random binary sequence and decode it using the codewords associated to the dictionary. Can you manage longer phrases so as to generate even more correct looking texts?

6. **Compression of DNA sequencing data**

   Next generation sequencing (NGS) platforms are generating massive amounts of short DNA reads in hundreds of laboratories around the world. The data contains lots of redundancy since the study targets are typically model organisms, whose whole genome sequences are almost completely known.

   The goal of the project is to develop a tailored compression scheme for short read data. The compression mechanism can exploit the reference genome for free (one can assume that both the compressor and decompressor have the same reference sequence available). The compression strategy can be arbitrary, but as a hint, it may be a good approach to align the reads to the reference and store somehow the occurrence positions and the list of edit differences.

   Reads are typically stored together with their encoded quality values: the sequencing machine associates to each position a quality value telling how likely the nucleotide at that position is correctly measured. For this project, we ignore the quality values. It is hence sufficient that the decompressor outputs the original read file without the quality values. Also each read has typically some (redundant) header information which can be ignored for this project.

   An example of input file for the compressor in fastq-format (`http://www.ncbi.nlm.nih.gov/sra/ERX001170?report=full`):

```
@ERR006459.1 091002_HWUSI-EAS451_0001_42FK7AAXX_K:3:1:0:1915 length=51
NAGAGAAAGAAGGAACCCTCCCTAAATCATTCCATGAAGCCAGTATCACCC
+ERR006459.1 091002_HWUSI-EAS451_0001_42FK7AAXX_K:3:1:0:1915 length=51
!IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIF?5II8>IH9@I/:4-.
@ERR006459.2 091002_HWUSI-EAS451_0001_42FK7AAXX_K:3:1:0:141 length=51
NTGGCGCATTTAAAGTAAGTGTGTGCAGAGACCAAGCCAAATGAGGCCCCA
+ERR006459.2 091002_HWUSI-EAS451_0001_42FK7AAXX_K:3:1:0:141 length=51
!EII$II2IIIIIII*IIIII3IHI.II5@G6+//-#(?$,.(5'*%#*(+
@ERR006459.3 091002_HWUSI-EAS451_0001_42FK7AAXX_K:3:1:0:95 length=51
NATCTCCCTGTTAGTGTCTTAAAAAAATCACAATAAAATATGTTGAAATTA
+ERR006459.3 091002_HWUSI-EAS451_0001_42FK7AAXX_K:3:1:0:95 length=51
!IIIIDDIIIIIIIIIIIIIIIIIIIIIDI<<I>I66/392<-495//.69
...
```

   An example of what the decompressor should output:

```
NAGAGAAAGAAGGAACCCTCCCTAAATCATTCCATGAAGCCAGTATCACCC
NTGGCGCATTTAAAGTAAGTGTGTGCAGAGACCAAGCCAAATGAGGCCCCA
NATCTCCCTGTTAGTGTCTTAAAAAAATCACAATAAAATATGTTGAAATTA
...
```

Notice that keeping the order for the reads is usually essential (if one stores headers and qualities compressed separately). Also for *paired-end* reads the order is essential: The reads are then stored in two files (usually named `*_1.fastq` and `*_2.fastq`) forming pairs of reads (from the same lines) such that one read represents the head and the other read represents the tail of a DNA fragment. The length of the DNA fragment is approximately known and hence the read pair should occur at specific distance from each others in the reference genome.

Therefore the compressor and decompressor should support the following modes for compression:

(a) Compressor gets one file at the time. Decompressor outputs the reads in the original order.

(b) Compressor gets one file at the time. Decompressor outputs the reads in arbitrary order.

(c) Compressor gets two files at the time (paired-end files). Decompressor outputs the reads in the original order.

(d) Compressor gets two files at the time (paired-end files). Decompressor outputs the reads in arbitrary order, yet synchronized between the two files.

Compression efficiency, compression time, and decompression time are obvious measures that sould be optimized in this project. A more advanced objective is to provide random access to compressed file, that is, to allow local decompression. In the case of short reads, the typical operation would be to extract the i-th read. Providing such functionality is considered voluntary in this project.

It is perfectly ok to use outside software to do parts of the compressor (but not the whole thing of course). For aligning the reads to the reference one can use e.g. `bwa`[1], `bowtie`[2], `SOAP2`[3], or `readaligner`[4]. For coding the occurrence positions, one can use integer codes such as implemented in `Integer Coding Library`[5].

7. **Compressing a Rank/Select Index**

Two of the most important operations for compressed and succinct data structures are rank(S, c, i) and select(S, c, i). Both functions operate on a sequence (string) $S[1, n]$ of n symbols. The symbols are drawn from an alphabet σ, so there are σ distinct symbols in S.

- rank(S, c, i): finds the number of occurrences of symbol c before position i, and
- select(S, c, i): finds the position of the ith occurrence of symbol c in S.

There are many data structures known for implementing these two operations, and some of the data structures use space proportional to that of the compressed size

[1]`http://bio-bwa.sourceforge.net/`

[2]`http://bowtie-bio.sourceforge.net/index.shtml`

[3]`http://soap.genomics.org.cn/soapaligner.html`

[4]`http://www.cs.helsinki.fi/group/suds/readaligner/`

[5]`http://www.di.unipi.it/~ferragin/Libraries/IntegerCoding/index.html`

of S. One of the fastest known data structures for rank/select is due to Golinski et al.[6,7], however it does not operate in compressed space.

In this project you will investigate whether Golinski's data structure becomes compressible when it is built for a particular type of string: the Burrows-Wheeler transform of S.

You will make use of an implementation of Golinski's data structure in Francisco Claude's LIBDCS library[8]. To construct BWT strings use the suffix sorting code by Yuta Mori[9]. For test data you can use the files available at Pizza Chili[10]. First use this implementation to gather statistics on the blocks used in Golinski's data structure. Then, devise a way to exploit any sparsity in the types of blocks which occur, and modify Claude's implementation accordingly.

Compare the practical performance of the modified data structure to the original implementation, in terms of both runtime (for rank and select) and space usage.

8. **Is the Longest Previous Factor table compressible?**

   Given a string $S[1, n]$, the Longest Previous Factor table stores, for each position i in S, a pair (j, ℓ), such that $S[i, i + \ell]$ is the longest substring starting at i in S which also occurs at some previous position $j < i$ in S.

   For example, the LPF pairs for string *abaabab* are $(-, 0), (-, 0), (1, 1), (1, 3), (2, 1)$. Often the pairs are thought of as two tables POS $= (-, -, 1, 1, 2)$ and LEN $= (0, 0, 1, 3, 1)$. LPF information is useful in a number of string processing applications, including compression (especially LZ77 based compression).

   Your task in this project is to determine the compressibility of the POS and LEN arrays. Start by exploring if compression is possible *in principle* using off the shelf compression tools (like 7zip) to try to compress POS and LEN. Keep in mind that it *might* help to transform POS and LEN in some (reversible) way before trying to compress them.

   To generate test data, implement one of the many algorithms for computing arrays POS and LEN have been published in recent years[11]. How does the compression of POS and LEN compare to the compression acheiveable on S, the original input string?

   Next, devise a more systematic method to compress POS and LEN while supporting random access to the original contents in a reasonable time. Implement your data structure.

[6] `http://dx.doi.org/10.1145/1109557.1109599`
[7] `http://www.dcc.uchile.cl/~gnavarro/ps/spire08.1.pdf`
[8] `http://libcds.recoded.cl/`
[9] `http://code.google.com/p/libdivsufsort/`
[10] `http://pizzachili.dcc.uchile.cl/`
[11] `http://dx.doi.org/10.1007/978-3-642-10217-2_18`

9. **Fixed Block Compression of Repetitive Data and Read Data**

   Fixed block compression is a simple and powerful technique for compressing string which have undergone the Burrows-Wheeler Transform (BWT). It was recently shown that this method leads to theoretically optimal text indexes that also perform well in practice[12]. The idea is to take a string T (the output of the BWT), divide it into block of equal size, and then compress each block with a compressor acheiving zero-order entropy (for example, using a Huffman coder).

   In this project you will assess the performance of fixed block compression on highly repetitive data[13], and on DNA read data (see project 6 above). You will need to compute BWTs of your test data, and to do this you should use Yuta Mori's suffix sorting code[14]. You will also need a Huffman coder to compress blocks[15].

   Experiment with different block sizes to produce graphs plotting compression as a function of block size. Compare your results to more complex methods designed for repetitive collections[16].

[12] `http://dx.doi.org/10.1007/978-3-642-24583-1_18`
[13] `http://pizzachili.dcc.uchile.cl/repcorpus.html`
[14] `http://code.google.com/p/libdivsufsort/`
[15] `http://ww2.cs.mu.oz.au/~alistair/mr_coder/`
[16] `http://dx.doi.org/10.1007/978-3-642-02008-7_9`