ParallelBWTzip

1.1

Generated by Doxygen 1.8.13

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 ActivePoint Struct Reference

`#include <suffixTree.h>`

Data Fields

- Node * activeNode
  Current active node.
- int activeLen
  Current active length.
- int activeEdge
  Current active edge.
- Node * root
  Root of the suffix tree.
- int * phase
  Current phase.

3.1.1 Detailed Description

This structure is used to keep track of the current active node, the current active edge and the current active length in order to speed up the construction of the suffix tree in the Ukkonen's version.
It also stores the pointer to the root of the tree and the current phase of the construction.

See also

`https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/`

Definition at line 113 of file suffixTree.h.

The documentation for this struct was generated from the following file:

- src/headers/suffixTree.h

3.2 Buffer Struct Reference

`#include <parallelBwtZip.h>`

Data Fields

- Queue * queue
  The shared queue.
- pthread_mutex_t mutex
  The mutex of the queue.
- pthread_cond_t cond
  The condition for the empty queue.

3.2.1 Detailed Description

Shared buffer data structure to implement the consumer-producer pattern to make the thread communicates.

Definition at line 66 of file parallelBwtZip.h.

The documentation for this struct was generated from the following file:

- src/headers/parallelBwtZip.h

3.3 ByteBuffer Struct Reference

`#include <arith.h>`

Data Fields

- unsigned char buf
  Byte buffer.
- unsigned bufBits
  Keeps track of the number of bits written.

3.3.1 Detailed Description

Store a byte bit by bit before it is printed out in the output buffer.

Definition at line 115 of file arith.h.

The documentation for this struct was generated from the following file:

- src/headers/arith.h

3.4 Decoder Struct Reference

`#include <unarith.h>`

Data Fields

- unsigned value
  Currently read code value to decompress.
- unsigned fin
  Number of bits passed the end of file.
- unsigned low
  Low value of the current interval.
- unsigned high
  High value of the current interval.
- unsigned range
  Range = High - Low.

3.4.1 Detailed Description

Maintains the state of the decoder.

Definition at line 50 of file unarith.h.

The documentation for this struct was generated from the following file:

- src/headers/unarith.h

3.5 Elem Struct Reference

`#include <queue.h>`

Data Fields

- Text elem
  One element of the queue.
- struct Elem * next
  Pointer to the next element in the queue.

3.5.1 Detailed Description

Linked list for the queue.

Definition at line 43 of file queue.h.

The documentation for this struct was generated from the following file:

- src/headers/queue.h

3.6 Encoder Struct Reference

`#include <arith.h>`

Data Fields

- unsigned low
  Low value of the current interval.
- unsigned high
  High value of the current interval.
- unsigned range
  Range = High - Low.
- unsigned underflow
  Keeps track of the underflow bits to be printed out.

3.6.1 Detailed Description

Maintains the state of the encoder, defined by the pair (low, high) that represents the border of the current interval of the compression. At the end of the compression any value between this interval will be ok to use for the arithmetic compression.

Definition at line 92 of file arith.h.

The documentation for this struct was generated from the following file:

- src/headers/arith.h

3.7 HashChildren Struct Reference

`#include <suffixTree.h>`

Data Fields

- unsigned firstChar
- Node * node
- UT_hash_handle hh

3.7.1 Detailed Description

This structure is used to maintain the list of children of each node. They are maintain in a table, created by using the library offered by Uthash.
For a node of the suffix tree, each suffix starting from it starts with a different letter. Thanks to this property the hash table that uses as key the firstChar of the edge that connects the child to the current node will have no collision.

See also

`https://troydhanson.github.io/uthash/`

Definition at line 73 of file suffixTree.h.

3.7.2 Field Documentation

3.7.2.1 firstChar

```
unsigned HashChildren::firstChar
```

Key value. It represents the first char of the suffix starting at the current node and terminating at the corresponding child.

Definition at line 75 of file suffixTree.h.

3.7.2.2 hh

```
UT_hash_handle HashChildren::hh
```

Makes this structure an hash table.

Definition at line 80 of file suffixTree.h.

3.7.2.3 node

```
Node* HashChildren::node
```

Child. Pointer to the corresponding child node.

Definition at line 78 of file suffixTree.h.

The documentation for this struct was generated from the following file:

- src/headers/suffixTree.h

3.8 Interval Struct Reference

```
#include <arith.h>
```

Data Fields

- unsigned lowCount
  Low value of the interval.
- unsigned highCount
  High value of the interval.
- unsigned scale
  Scale.

3.8.1 Detailed Description

The counts uniquely define where on the 0 to 1 range the symbol lies, and the scale value tells what the total span of the 0 to 1 range scale is.

If 1000 characters are read so far, scale will be 1000.

To calculate the current interval we calculate: (lowCount/scale, highCount/scale)

Definition at line 78 of file arith.h.

The documentation for this struct was generated from the following file:

- src/headers/arith.h

3.9 IOBuffer Struct Reference

`#include <arith.h>`

Data Fields

- unsigned char * text

  Maintains the already written compressed text.
- int index

  Index of the current char to be written.

3.9.1 Detailed Description

Buffer that stores the intermediate results of the compression. Each time the byte buffer is full, it is emptied into the IOBuffer.

Definition at line 126 of file arith.h.

The documentation for this struct was generated from the following file:

- src/headers/arith.h

3.10 Model Struct Reference

`#include <arith.h>`

Data Fields

- unsigned size

  Number of probabilities stored in.
- unsigned short * freq

3.10.1 Detailed Description

Stores the model of probabilities. It maintains 256 cumulative probabilities one for each possible character in input between 0 and 255.

Definition at line 105 of file arith.h.

3.10.2 Field Documentation

3.10.2.1 freq

```
unsigned short* Model::freq
```

Array of the cumulative probabilities of each value of the mdoel.

Definition at line 108 of file arith.h.

The documentation for this struct was generated from the following file:

- src/headers/arith.h

3.11 Node Struct Reference

```
#include <suffixTree.h>
```

Data Fields

- int start
- int * end
- int suffixIndex
- HashChildren * children
- Node * suffixLink

3.11.1 Detailed Description

In the node structure there is also information about the label of the edge that connects it to its parent. In particular the suffix on each edge is encoded in the child node of the edge by storing the indexes that delimit the suffix in the input string.

Definition at line 90 of file suffixTree.h.

3.11.2 Field Documentation

3.11.2.1 children

```
HashChildren* Node::children
```

Pointer to the hash table containing the children of the node. NULL if it is a leaf.

Definition at line 98 of file suffixTree.h.

3.11.2.2 end

```
int* Node::end
```

Pointer to the index of the end of the suffix in the edge with its parent.

Definition at line 94 of file suffixTree.h.

3.11.2.3 start

```
int Node::start
```

Index of the start of the suffix in the edge with its parent.

Definition at line 92 of file suffixTree.h.

3.11.2.4 suffixIndex

```
int Node::suffixIndex
```

Suffix index of the nodes. (-1 if the node is internal).

Definition at line 96 of file suffixTree.h.

3.11.2.5 suffixLink

```
Node* Node::suffixLink
```

Pointer to another node in a different subtree.

Definition at line 100 of file suffixTree.h.

The documentation for this struct was generated from the following file:

- src/headers/suffixTree.h

3.12 Queue Struct Reference

`#include <queue.h>`

Data Fields

- Elem * front
- Elem * rear

  Pointer to the last element of the queue.

3.12.1 Detailed Description

Queue. The counter counts how many elements enter in the queue, but then it does not decrement the counter. So it just counts how many element are entered into the queue.

Definition at line 55 of file queue.h.

3.12.2 Field Documentation

3.12.2.1 front

`Elem* Queue::front`

Stores the number of element entered in the queue. Pointer to the first element to be filled in of the queue.

Definition at line 59 of file queue.h.

The documentation for this struct was generated from the following file:

- src/headers/queue.h

3.13 SymbolsList Struct Reference

`#include <mtf.h>`

Data Fields

- unsigned char symbol

  One symbol.
- struct SymbolsList * next

  Pointer to the next symbol in the list.

3.13.1 Detailed Description

A list of symbols, from 0 to 255.

Definition at line 54 of file mtf.h.

The documentation for this struct was generated from the following file:

- src/headers/mtf.h

3.14 Text Struct Reference

`#include <util.h>`

Data Fields

- unsigned char * text

  Pointer to the sequence of bytes.
- size_t len

  The length of the sequence of bytes.
- long id

  The id of the chunk size.

3.14.1 Detailed Description

This structure is the one who stores the sequence of bytes to be compress or decompress together with its length. In addition, the id variable stores the id of the chunk during compression. This is foundamental for the parallel version of the compression algorithm, since the execution of the chunk compression could be out of order, so they need to be ordered at the end of the compression.

Definition at line 57 of file util.h.

The documentation for this struct was generated from the following file:

- src/headers/util.h

Chapter 4

File Documentation

4.1 src/headers/arith.h File Reference

Header file implementing an arithmetic coder compressor.

```
#include "util.h"
```

Data Structures

- struct Interval
- struct Encoder
- struct Model
- struct ByteBuffer
- struct IOBuffer

Typedefs

- typedef struct Interval Interval
- typedef struct Encoder Encoder
- typedef struct Model Model
- typedef struct ByteBuffer ByteBuffer
- typedef struct IOBuffer IOBuffer

Enumerations

- enum FREQUENCY { FREQUENCY_BITS = 15, MAX_FREQUENCY = POW_2(FREQUENCY_BITS) - 1 }
- enum BITS { BITS_RANGE = 17, MAX_RANGE = POW_2(BITS_RANGE) }

Functions

- Text encodingRoutine (const Text input)

  Main function of the arithmetic encoder.
- void initEncoder (Encoder *const encoder)

  Initialize the encoder pointed by the pointer in input.
- void encodeSymbol (unsigned *const low, unsigned *const high, const unsigned range, const Interval interval)

  Encode the symbol by scaling the range basing on the interval of the symbol.
- void findInterval (Interval *const interval, Model *const model, const unsigned ch)

  Find the interval probability of the input symbol.
- void finishEncoding (Encoder *const encoder, ByteBuffer *const byteBuf, IOBuffer *const buf)

  It checks if there are still bits to write in output, and writes it.
- void initModel (Model *const model, const unsigned size)

  Initialization of the mdoel. Assigns a uniform probability to all the values.
- void updateModel (Model *const model, const unsigned ch)

  Updates the model basing on the new occurrence of the character in input.
- void initByteBuffer (ByteBuffer *byteBuf)

  Byte buffer initialization.
- void checkForOutputBit (Encoder *const encoder, ByteBuffer *const byteBuf, IOBuffer *const buf)

  Check if there are bits that can be output and it output those bit in the byte buffer.
- void outputBits (Encoder *const encoder, const unsigned bit, ByteBuffer *const byteBuf, IOBuffer *const buffer)

  Output the bit in input in the byte Buffer and checks and outputs underflow bits.
- void outputBit (ByteBuffer *const byteBuf, const unsigned bit, IOBuffer *const buffer)

  Writes the bit in the byte buffer and then, if the byte buffer is full, it empties it into the buffer.

4.1.1 Detailed Description

Header file implementing an arithmetic coder compressor.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the function needed to perform an airthmetic compression, that is the last phase of the BWT compression. The code has been taken from https://github.com/wonder-mice/arcd even is it is slightly modified to make it fit better in the application.
The header offers the possibility to encode a given sequence of bytes in input, by using an adaptive model of probabilities. It also gives a set of functions to print out the encode sequence byte by byte.
The parameters for the compression such as the number of bits to represent probability intervals are fixed because the purpose of the application was not to found the best compression possible, rather it aims to have an optimal compression time with a good compression rate.

4.1.2 Typedef Documentation

4.1.2.1 ByteBuffer

```
typedef struct ByteBuffer ByteBuffer
```

Store a byte bit by bit before it is printed out in the output buffer.

4.1.2.2 Encoder

```
typedef struct Encoder Encoder
```

Maintains the state of the encoder, defined by the pair (low, high) that represents the border of the current interval of the compression. At the end of the compression any value between this interval will be ok to use for the arithmetic compression.

4.1.2.3 Interval

```
typedef struct Interval Interval
```

The counts uniquely define where on the 0 to 1 range the symbol lies, and the scale value tells what the total span of the 0 to 1 range scale is.

If 1000 characters are read so far, scale will be 1000.

To calculate the current interval we calculate: (lowCount/scale, highCount/scale)

4.1.2.4 IOBuffer

```
typedef struct IOBuffer IOBuffer
```

Buffer that stores the intermediate results of the compression. Each time the byte buffer is full, it is emptied into the IOBuffer.

4.1.2.5 Model

```
typedef struct Model Model
```

Stores the model of probabilities. It maintains 256 cumulative probabilities one for each possible character in input between 0 and 255.

4.1.3 Enumeration Type Documentation

4.1.3.1 BITS

```
enum BITS
```

Number of bits used to represent arithmetic coder intervals. This value is not exact, in fact the maximum value uses one bit more, but it's the only one.

Enumerator

BITS_RANGE	Number of bits.
MAX_RANGE	Maximum possible range.

Definition at line 61 of file arith.h.

4.1.3.2 FREQUENCY

```
enum FREQUENCY
```

Defines how to represent frequency values. This value is exact.

Enumerator

FREQUENCY_BITS	Number of bits.
MAX_FREQUENCY	Max value of bits.

Definition at line 50 of file arith.h.

4.1.4 Function Documentation

4.1.4.1 checkForOutputBit()

```
void checkForOutputBit (
            Encoder *const encoder,
            ByteBuffer *const byteBuf,
            IOBuffer *const buf )
```

Check if there are bits that can be output and it output those bit in the byte buffer.

Parameters

in,out	*encoder*	The encoder.
in,out	*byteBuf*	The byte buffer.
in,out	*buf*	The buffer for output.

Definition at line 202 of file arith.c.

4.1.4.2 encodeSymbol()

```
void encodeSymbol (
            unsigned *const low,
            unsigned *const high,
            const unsigned range,
            const Interval interval )
```

Encode the symbol by scaling the range basing on the interval of the symbol.

Parameters

in,out	*low*	Current low value of the interval.
in,out	*high*	Current high value of the interval.
in	*range*	Current range.
in	*interval*	Interval relative to the current symbol in input.

Definition at line 118 of file arith.c.

4.1.4.3 encodingRoutine()

```
Text encodingRoutine (
            const Text input )
```

Main function of the arithmetic encoder.

Parameters

in	*input*	A sequence of bytes.

Returns

The compressed text.

This functions compresses an input sequence of bytes using an arithmetic encoder and an adaptive model of probabilities.
It reads the sequence byte by byte and for each byte:

- Encode the symbol using the actual probabilities.
- Check if the ByteBuffer is full to print out a char.
- Update probabilities.

Definition at line 51 of file arith.c.

4.1.4.4 findInterval()

```
void findInterval (
            Interval *const interval,
            Model *const model,
            const unsigned ch )
```

Find the interval probability of the input symbol.

Parameters

out	*interval*	The interval that needs to be found.
in	*model*	The model of probabilities.
in	*ch*	The character which interval has to be found.

Definition at line 128 of file arith.c.

4.1.4.5 finishEncoding()

```
void finishEncoding (
            Encoder *const encoder,
            ByteBuffer *const byteBuf,
            IOBuffer *const buf )
```

It checks if there are still bits to write in output, and writes it.

Parameters

in	*encoder*	The encoder.
in,out	*byteBuf*	The byte buffer.
in,out	*buf*	The buffer for output.

Definition at line 140 of file arith.c.

4.1.4.6 initByteBuffer()

```
void initByteBuffer (
            ByteBuffer * byteBuf )
```

Byte buffer initialization.

Parameters

in,out	*byteBuf*	

Definition at line 195 of file arith.c.

4.1.4.7 initEncoder()

```
void initEncoder (
            Encoder *const encoder )
```

Initialize the encoder pointed by the pointer in input.

Parameters

in,out	*encoder*	The encoder is initialize so that the intervals is the total interval [0,1].

Definition at line 109 of file arith.c.

4.1.4.8 initModel()

```
void initModel (
            Model *const model,
            const unsigned size )
```

Initialization of the mdoel. Assigns a uniform probability to all the values.

Parameters

in,out	*model*	The model to initialize.
in	*size*	The size of the model, i.e. the number of probabilities.

Definition at line 164 of file arith.c.

4.1.4.9 outputBit()

```
void outputBit (
            ByteBuffer *const byteBuf,
            const unsigned bit,
            IOBuffer *const buffer )
```

Writes the bit in the byte buffer and then, if the byte buffer is full, it empties it into the buffer.

Parameters

in,out	*byteBuf*	The byte buffer.
in	*bit*	The bit to be printed.
in,out	*buffer*	The buffer.

Definition at line 245 of file arith.c.

4.1.4.10 outputBits()

```
void outputBits (
            Encoder *const encoder,
            const unsigned bit,
            ByteBuffer *const byteBuf,
            IOBuffer *const buffer )
```

Output the bit in input in the byte Buffer and checks and outputs underflow bits.

Parameters

in,out	*encoder*	The encoder.
in	*bit*	The bit to be printed out.
in,out	*byteBuf*	The byte buffer to store the current byte to be printed.
in,out	*buffer*	The buffer to store the current overall output.

Definition at line 233 of file arith.c.

4.1.4.11 updateModel()

```
void updateModel (
            Model *const model,
            const unsigned ch )
```

Updates the model basing on the new occurrence of the character in input.

Parameters

in,out	*model*	The model to be updated.
in	*ch*	The new occurrence for which the model has to be updated.

Definition at line 174 of file arith.c.

4.2 src/headers/bwt.h File Reference

Header file implementing the Burrows-Wheeler transformation (BWT).

```
#include "suffixTree.h"
```

Functions

- Text bwtTransformation (const Text input)

  Returns the BWT transformation of the input text.
- Text getBWT (unsigned ∗const input, const size_t inputLen, Node ∗const suffixTree)

  Returns the BWT transformation of the input syntax tree.
- void createSuffixArray (Node ∗const node, unsigned ∗const input, int ∗const i, int ∗const suffixArray)

  Creates the suffix array recursively, starting from the root of the suffix tree in input and frees the suffix tree.
- unsigned ∗ setSentinel (Text input)

  Set the sentinel at the end of the input Text returning a pointer to a sequence of unsigned.

4.2.1 Detailed Description

Header file implementing the Burrows-Wheeler transformation (BWT).

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the functions needed to obtain, given a sequence of bytes, the Burrows-Wheeler transformation of that sequence. By calling the bwtTransformation function pssing as argument the sequence of bytes, we obtain returned the bwt together with its length.
The function getBWT offers the possibility to obtain the BWT, by passing as argument the input sequence and the suffix tree of the sequence.
Finally the function createSuffixArray allows to create the suffix array of a sequence of bytes from the suffix tree of that sequence.

4.2.2 Function Documentation

4.2.2.1 bwtTransformation()

```
Text bwtTransformation (
            const Text input )
```

Returns the BWT transformation of the input text.

Parameters

in	*input*	Sequence of bytes together with its length.

Returns

The BWT of the input text.

This function return the BWT transformation of the input text in three phases:

- First set the sentinel character in the input text. This step is fundamental to obtain the suffix array, since the transformation is reversible if and only if the input text ends with a character different from all the others.
- Second the suffix tree is built.
- Finally the BWT is created by calling the function getBWT.

Definition at line 44 of file bwt.c.

4.2.2.2 createSuffixArray()

```
void createSuffixArray (
            Node *const node,
            unsigned *const input,
            int *const i,
            int *const suffixArray )
```

Creates the suffix array recursively, starting from the root of the suffix tree in input and frees the suffix tree.

Parameters

in	*node*	Current pointer to the node of the recursion.
in	*input*	Input sequence terminating with the sentinel char.
in,out	*i*	Current pointer to the index of the suffix array to be filled in.
in,out	*suffixArray*	Pointer to the suffix array to be created.

Lexicographic depth-first search traversal to create the suffix array. The suffix tree is recursively traversed and as soon as a node and all its children have been visited they are deleted.

Definition at line 116 of file bwt.c.

4.2.2.3 getBWT()

```
Text getBWT (
            unsigned *const input,
```

```
            const size_t inputLen,
            Node *const suffixTree )
```

Returns the BWT transformation of the input syntax tree.

Note that in this case the input text must terminate with the sentinel character. You can call the function setSentinel to get the input with the sentinel concatenated.

Parameters

in	*input*	Sequences of bytes with sentinel character at the end.
in	*inputLen*	Length of the input sequence.
in	*suffixTree*	Suffix tree of the input sequence.

Returns

The BWT of the syntax tree in input.

This function encodes the BWT in the output array of bytes. To make the transformation reversible we store in the output array also the index of the sentinel character and the index of the first character of the input. These two index can be bigger than a byte, so they are encoded in two bytes each.
The output array will have:

- First two bytes storing the encode unsigned index of the first char.
- Third and fourth byte storing the encode usigned index of the sentinel.
- The sentinle character itself is encoded as zero.

In this way we can maintain the result in an array of unsigned char.
The operation is performed in two steps:

- Create suffix array starting from the suffix tree.
- Extract the BWT from the suffix array.

Definition at line 73 of file bwt.c.

4.2.2.4 setSentinel()

```
unsigned* setSentinel (
            Text input )
```

Set the sentinel at the end of the input Text returning a pointer to a sequence of unsigned.

Parameters

in	*input*	Sequence of bytes.

Returns

A pointer to the sequence in input, terminating with the sentinel char.

This functions returns the input text with the sentinel character concatenated at the end. The pointer returned is unsigned so that it can contains also the sentinel character (i.e. char 256). Since the input text is of type unsigned char, we cna be sure that the sentinel is the only character in the text.

Definition at line 156 of file bwt.c.

4.3 src/headers/bwtUnzip.h File Reference

Header file implementing the sequential version of the Burrows-Wheeler decompression.

```
#include "parallelBwtZip.h"
#include "unbwt.h"
#include "unmtf.h"
#include "unzle.h"
#include "unarith.h"
```

Functions

- Text bwtUnzip (const Text input)

  Performs the BWT decompression.
- void decompress (FILE *input, FILE *output)

  Decompress a given input file, writing the result in the specified output file.

4.3.1 Detailed Description

Header file implementing the sequential version of the Burrows-Wheeler decompression.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This interface provides the functions needed to decompress a given file. The function bwtUnzip decompresses a given sequence of bytes and returns a decompressed sequence of bytes. While decompress decompresses an entire file, previously compressed. The file must respect the format defined in the compression version.

4.3.2 Function Documentation

4.3.2.1 bwtUnzip()

```
Text bwtUnzip (
            const Text input )
```

Performs the BWT decompression.

Parameters

in	*input*	The sequence of bytes to decompress.

Returns

The sequence of bytes decompressed.

This function applies in reverse the four step of the BWT decompression in sequence to a single sequence of bytes. The main phase are:

1. Arithmetic decoding.
2. Zero-length deconding.
3. MTF reverse transformation.
4. BWT reverse transformation.

Definition at line 38 of file bwtUnzip.c.

4.3.2.2 decompress()

```
void decompress (
            FILE * input,
            FILE * output )
```

Decompress a given input file, writing the result in the specified output file.

Parameters

in	*input*	The file to decompress
out	*output*	The file where decompress.

The compressed file is read. First the encoded length and the id are extract from the file. Then if the id is 1 the compressed chunk of the length read before is decompressed and written out in the output file. The code is consistency with the compression version.

Definition at line 61 of file bwtUnzip.c.

4.4 src/headers/mtf.h File Reference

Header file implementing the move to front transformation.

```
#include "util.h"
```

Data Structures

- struct SymbolsList

Typedefs

- typedef struct SymbolsList SymbolsList

Functions

- Text mtf (const Text input)

  Transform the input sequence of bytes into an array of output bytes, indices of the symbol list.
- int search (SymbolsList *const symbols, const unsigned char byte, SymbolsList **aux)

  Returns the position of the searched byte in the list of symbols, or -1 if the element is not found.
- SymbolsList * mvtElement (SymbolsList *const symbols, SymbolsList *el)

  Move the element pointed by el to the front of the list in input.
- SymbolsList * initListOfSymbols (void)
- void freeListOfSymbols (SymbolsList *symbols)

4.4.1 Detailed Description

Header file implementing the move to front transformation.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the functions needed to perform the move to front (MTF) transformation, that is the second phase of the BWT compression.
The algorithm takes an input byte and outputs the number of distinct other bytes that have been seen since its last occurrence. At the beginning the 0x00 byte is assumed to be the most recently occurred, with the other bytes in sequence.
The implementation is done by means of a list of 256 bytes, where the most recently seen bytes are kept in order from position 0. When a new byte is read, its position in the list is output and the byte is moved in front of the list. The output is still a sequence of bytes from 0 to 255.

Note

If the most recently byte is read, i.e. we have at least two consecutive equal characters, the algorithm outputs 0 for the second one.

4.4.2 Typedef Documentation

4.4.2.1 SymbolsList

```
typedef struct SymbolsList SymbolsList
```

A list of symbols, from 0 to 255.

4.4.3 Function Documentation

4.4.3.1 freeListOfSymbols()

```
void freeListOfSymbols (
            SymbolsList * symbols )
```

Parameters

in	*symbols*	Pointer to the head of the list to be freed.

Free the list of symbols and deallocate memory.

Definition at line 152 of file mtf.c.

4.4.3.2 initListOfSymbols()

```
SymbolsList* initListOfSymbols (
            void  )
```

Returns

The list of symbols initialized.

Populate the list of symbols with characters from 0 to 255

Definition at line 129 of file mtf.c.

4.4.3.3 mtf()

```
Text mtf (
            const Text input )
```

Transform the input sequence of bytes into an array of output bytes, indices of the symbol list.

Parameters

in	*input*	The input sequence of bytes together with its length.

Returns

The transformed array of bytes.

The function scan the input array of bytes and for each byte it performs three actions:

- Search the position of the input byte.
- Write the position in the output array.
- Move to the head of the symbol list the read byte.

Definition at line 37 of file mtf.c.

4.4.3.4 mvtElement()

```
SymbolsList* mvtElement (
             SymbolsList *const symbols,
             SymbolsList * el )
```

Move the element pointed by el to the front of the list in input.

This operation is performed only if the searched element was not at the head of the list.

Parameters

in,out	*symbols*	Pointer to the head of the list of symbols.
in	*el*	The element before the one to be moved in front of the list.

Returns

The current head of the list.

This function moves the element pointed by the one given in input to the front of the list of symbols. The function is carried out with two operations:

- Delete element from the list of symbols.
- Re-insert it at the head of the list.

Definition at line 112 of file mtf.c.

4.4.3.5 search()

```
int search (
            SymbolsList *const symbols,
            const unsigned char byte,
            SymbolsList ** aux )
```

Returns the position of the searched byte in the list of symbols, or -1 if the element is not found.

Parameters

in	*symbols*	Pointer to the head of the list of symbols.
in	*byte*	The byte to be searched in the list.
in,out	*aux*	Pointer to the previous element of the searched byte in the list, useful later to move ahead the returned element.

Returns

The position of the byte in the list of symbols. The function returns -1 (error) if the element is not found.

The function implements a simple search of an element in a list of bytes. It outputs the corresponding position of the element and, at the same time, it stores in the double pointer aux the element before the one searched in the list, so as to make the next operation of moving ahead the element quicker.

Definition at line 79 of file mtf.c.

4.5 src/headers/parallelBwtZip.h File Reference

Header file implementing the parallel version of the Burrows-Wheeler compression.

```
#include <pthread.h>
#include <unistd.h>
#include <math.h>
#include <sched.h>
#include "bwt.h"
#include "mtf.h"
#include "zle.h"
#include "arith.h"
#include "queue.h"
#include "util.h"
```

Data Structures

- struct Buffer

Macros

- #define NUM_THREADS 8

  Number of threads.

Typedefs

- typedef struct Buffer Buffer

Functions

- void compressParallel (FILE *input, FILE *output, const long chunkSize)

  Performs the parallel BWT compression.
- void * bwtStage (void *)

  Thread for the Burrow-Wheeler transform stage.
- void * mtfZleStage (void *)

  Thread for the MTF and ZLE stage.
- void * arithStage (void *)

  Thread for the arithmetic coding stage.
- void initBuffer (Buffer *const buf)

  Initialize a queue buffer.
- void writeOutput (FILE *output, int *const index, const int littleChunk)

  Write the compressed output from the result array to the output file.
- void setAffinity (cpu_set_t *const cpus, int cpu, pthread_attr_t *const attr)

  Set a specific core where a thread will be run.

4.5.1 Detailed Description

Header file implementing the parallel version of the Burrows-Wheeler compression.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This interface defines the threads and the functions needed to implement the BWT compression in parallel.
The four task of the BWT algorithm has been divided into three set:

- One for the Burrows-Wheeler transformation.
- One for the move to front transformation and the zero-length encoding.
- One for the arithmetic coding compression.

The main thread reads the file chunk by chunk and outputs the compressed chunks as soon as they become available.
The header offers also a function to set a particular logical core to a thread.

4.5.2 Typedef Documentation

4.5.2.1 Buffer

```
typedef struct Buffer Buffer
```

Shared buffer data structure to implement the consumer-producer pattern to make the thread communicates.

4.5.3 Function Documentation

4.5.3.1 arithStage()

```
void* arithStage (
            void * arg )
```

Thread for the arithmetic coding stage.

Parameters

arg	NULL.

Performs the bwt on a chunk of data arith from the read buffer as soon it is available. Note that the thread must first lock the mutex on the arith buffer and checks that it is not empty. Then it can read the data and perform the operation. Finally it needs to lock the lock on the output array, in this case the result array, before written the result into the array.

Definition at line 285 of file parallelBwtZip.c.

4.5.3.2 bwtStage()

```
void* bwtStage (
            void * arg )
```

Thread for the Burrow-Wheeler transform stage.

Parameters

arg	NULL.

Performs the bwt on a chunk of data read from the read buffer as soon it is available. Note that the thread must first lock the mutex on the read buffer and checks that it is not empty. Then it can read the data and perform the

operation. Finally it needs to lock the lock on the output buffer, in this case the bwt buffer, before written the result into the buffer.

Definition at line 202 of file parallelBwtZip.c.

4.5.3.3 compressParallel()

```
void compressParallel (
            FILE * input,
            FILE * output,
            const long chunkSize )
```

Performs the parallel BWT compression.

Parameters

in	*input*	The input file to compress.
in,out	*output*	The output where to write the compressed file.
in	*chunkSize*	The size of the chunks in which the file must be splitted before compression.

This function applies the BWT compression to a file in input. The main thread creates other 8 threads: 6 for the BWT phase, 1 for the MTF and ZLE phases and one for the arithmetic phase. Threads communicate with each other by using shared global buffers. Each thread, including the main one, acts as consumer as well as a producer. In fact:

- The main thread is the producer of the read buffer because it fills in it after the read of one chunk of bytes from the input file. Then it acts as a consumer of the result array, because it has to write the compressed chunk store in the result in the output file.
- The bwt thread is the consumer of the read buffer and the producer of the bwt buffer.
- The mtf and zle thread is the consumer of the bwt buffer and the producer of the arith buffer.
- Finally the arith thread is the consumer of the arith buffer and the producer of the result array.

Each chunk of data is processed asynchronously and they are written in order given the id of the chunk, setted before the compression starts.

Definition at line 91 of file parallelBwtZip.c.

4.5.3.4 initBuffer()

```
void initBuffer (
            Buffer *const buf )
```

Initialize a queue buffer.

Parameters

in,out	*buf*	The buffer to be initialized.

Definition at line 316 of file parallelBwtZip.c.

4.5.3.5 mtfZleStage()

```
void* mtfZleStage (
            void * arg )
```

Thread for the MTF and ZLE stage.

Parameters

arg	NULL.

Performs the bwt on a chunk of data read from the bwt buffer as soon it is available. Note that the thread must first lock the mutex on the bwt buffer and checks that it is not empty. Then it can read the data and perform the operation. Finally it needs to lock the lock on the output buffer, in this case the arith buffer, before written the result into the buffer.

Definition at line 243 of file parallelBwtZip.c.

4.5.3.6 setAffinity()

```
void setAffinity (
            cpu_set_t *const cpus,
            int cpu,
            pthread_attr_t *const attr )
```

Set a specific core where a thread will be run.

Parameters

in	*cpus*	Pointer to the number of cpus.
in,out	*cpu*	The cpu number to be set for the thread.
in,out	*attr*	Pointer to the attribute of the thread that must run on the specific core.

Definition at line 356 of file parallelBwtZip.c.

4.5.3.7 writeOutput()

```
void writeOutput (
            FILE * output,
            int *const index,
            const int littleChunk )
```

Write the compressed output from the result array to the output file.

Parameters

in	*output*	Pointer to the result to be written out.
in,out	*index*	Pointer to the index of the next cell of the output array to be written out.
in	*littleChunk*	0 if the chunk has not been compressed, 1 otherwise

The main thread continuously calls this function to check if the next block to be output is available.
The index keeps trace of the id of the next block to output. The id of the result are initialized to -1 so the main thread knows when a compressed block has been stored into the array.

Definition at line 329 of file parallelBwtZip.c.

4.6 src/headers/queue.h File Reference

Header file implementing a queue.

```
#include "util.h"
```

Data Structures

- struct Elem
- struct Queue

Typedefs

- typedef struct Elem Elem
- typedef struct Queue Queue

Functions

- void initQueue (Queue *q)

  Queue initialization.
- void enqueue (Text elem, Queue *q)

  Adds an element at the front of the queue.
- Text dequeue (Queue *q)

  Deletes an element from the rear of the queue and returns it.
- int empty (Queue *q)

  Checks if the queue is empty.

4.6.1 Detailed Description

Header file implementing a queue.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the function to implement a queue of Text using a linked list. The Queue data structure has two pointer for the front and the rear of the queue, respectively. In addition it has also a counter for counting the number of element in the queue.

4.6.2 Typedef Documentation

4.6.2.1 Elem

```
typedef struct Elem Elem
```

Linked list for the queue.

4.6.2.2 Queue

```
typedef struct Queue Queue
```

Queue. The counter counts how many elements enter in the queue, but then it does not decrement the counter. So it just counts how many element are entered into the queue.

4.6.3 Function Documentation

4.6.3.1 dequeue()

```
Text dequeue (
            Queue * q )
```

Deletes an element from the rear of the queue and returns it.

Parameters

in,out	*q*	The queue where the element must be retrieved.

Returns

The element dequeued.

Definition at line 58 of file queue.c.

4.6.3.2 empty()

```
int empty (
            Queue * q )
```

Checks if the queue is empty.

Parameters

in	*q*	The queue.

Returns

- 1 if the queue is empy.
- 0 otherwise.

Definition at line 74 of file queue.c.

4.6.3.3 enqueue()

```
void enqueue (
            Text elem,
            Queue * q )
```

Adds an element at the front of the queue.

Parameters

in	*elem*	The element to be added.
in,out	*q*	The queue qhere the element must be added.

Definition at line 41 of file queue.c.

4.6.3.4 initQueue()

```
void initQueue (
            Queue * q )
```

Queue initialization.

Parameters

in,out	*q*	The queue to initialize.

Definition at line 33 of file queue.c.

4.7 src/headers/sequentialBwtZip.h File Reference

Header file implementing the sequential version of the Burrows-Wheeler compression.

```
#include "bwt.h"
#include "mtf.h"
#include "zle.h"
#include "arith.h"
```

Functions

- Text bwtZip (const Text input)
  Performs the sequential BWT compression.
- void compressSequential (FILE *input, FILE *output, long chunkSize)
  Compress a given input file, writing the result in the specified output file.

4.7.1 Detailed Description

Header file implementing the sequential version of the Burrows-Wheeler compression.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This interface provides the functions needed to compress a given file. The function bwtZip compressed a given sequence of bytes and returns a compressed sequence of bytes. While compressSequential compresses an entire file, divided into chunks of some specified size.

4.7.2 Function Documentation

4.7.2.1 bwtZip()

```
Text bwtZip (
            const Text input )
```

Performs the sequential BWT compression.

Parameters

in	*input*	The sequence of bytes to compress.

Returns

The compressed sequence of bytes.

This function applies the four step of the BWT compression in sequence to a single sequence of bytes.
The main phase are:

1. BWT transformation.
2. MTF transformation.
3. Zero-length encoding.
4. Arithmetic compression.

Definition at line 38 of file sequentialBwtZip.c.

4.7.2.2 compressSequential()

```
void compressSequential (
            FILE * input,
            FILE * output,
            long chunkSize )
```

Compress a given input file, writing the result in the specified output file.

Parameters

in	*input*	The file to compress
out	*output*	The file where compress.
in	*chunkSize*	The size of the chunks in which the file must be splitted for compression.

The file is read chunk by chunk until is finished. Each chunk is compressed when is read and then written in output

folloring its length (encoded in 4 bytes) and the id of the chunk. If one chunk is smaller than MIN_CHUNK_SIZE the chunk is not compressed for efficiency reasons.
The id is 1 if the chunk is compressed, 0 otherwise.

Definition at line 63 of file sequentialBwtZip.c.

4.8 src/headers/suffixTree.h File Reference

Header file implementing the construction of the suffix tree given a sequence of bytes.

```
#include "../extern/uthash.h"
#include "util.h"
```

Data Structures

- struct HashChildren
- struct Node
- struct ActivePoint

Typedefs

- typedef struct Node Node
- typedef struct HashChildren HashChildren
- typedef struct ActivePoint ActivePoint

Functions

- Node * buildSuffixTree (unsigned *const input, const size_t inputLen)

  Returns the root of the suffix tree of the input sequence of bytes.
- void addSuffixIndex (Node *const node, const int labelLen, const size_t inputLen)

  Add the suffix indexes to the leaves of the suffix tree.
- void applyExtensions (unsigned *const input, int *const remainder, ActivePoint *const ap)

  Apply the extensions to the tree.
- Node * createNode (const int start, int *const end, Node *const root)

  Creates a node with default initialization.
- Node * createInternalNode (ActivePoint *const ap, HashChildren **const child, unsigned *const input)

  Splits an edge by creating a new internal and a new leaf.
- void createLeaf (ActivePoint *const ap, Node *const parent, unsigned *const input)

  Creates a leaf of the tree and attach it to the parent passed as arg.
- void addNewChild (Node *const child, Node *const parent, unsigned *const input)

  Attachs a new child to a node.
- void setSuffixLink (Node *const node, Node **newest)

  Sets a suffix link to a node.
- void updateAP (ActivePoint *const ap, const int remainder)

  Update the active point.
- int walkDown (ActivePoint *const ap, Node *const currentNode)

  Walk down the tree onw level following one edge.

- int getEdgeLen (Node *const node)

  Returns the length of the label on the edge terminating in the input node.
- void deleteNode (Node *node)

  Delete a node by free the memory for it and all of its children.
- void deleteChildren (HashChildren **children)

  Frees the hash table that contains the children of a node.
- void printTree (Node *node, unsigned *input)

  Depth-first traversal of the tree to print it.

4.8.1 Detailed Description

Header file implementing the construction of the suffix tree given a sequence of bytes.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the function needed to build the suffix tree of a sequence of bytes. The most relevant functions of the interface are buildSuffixTree and addSuffixIndex. The last one is used to add the suffix index to the leaves of the tree. They can be useful to extract properties of the sequence in input.
The interface offers also the possibility to delete one node (deleteNode) and to print the tree (printTree).
The other functions are used for the construction fo the tree. The algorithm implements the online construction of suffix tree introduced by Ukkonen. The implementation follows the one proposed by http://www.geeksforgeeks.org.

See also

https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/
E. Ukkonen. “On-line construction of suffix trees”, Algorithmica, 14(3):249-260, 1995.

4.8.2 Typedef Documentation

4.8.2.1 ActivePoint

```
typedef struct ActivePoint ActivePoint
```

This structure is used to keep track of the current active node, the current active edge and the current active length in order to speed up the construction of the suffix tree in the Ukkonen's version.
It also stores the pointer to the root of the tree and the current phase of the construction.

See also

https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/

4.8.2.2 HashChildren

```
typedef struct HashChildren HashChildren
```

This structure is used to maintain the list of children of each node. They are maintain in a table, created by using the library offered by Uthash.
For a node of the suffix tree, each suffix starting from it starts with a different letter. Thanks to this property the hash table that uses as key the firstChar of the edge that connects the child to the current node will have no collision.

See also

https://troydhanson.github.io/uthash/

4.8.2.3 Node

```
typedef struct Node Node
```

The node structure is composed by several members that can be divided in two groups:

- Three values to store the start and end indexes of the suffix in the edge connecting it to its parent, and its suffixIndex.
- The links with its children and the suffix link, that is used in the construction of the tree proposed by Ukkonen.

Definition at line 61 of file suffixTree.h.

4.8.3 Function Documentation

4.8.3.1 addNewChild()

```
void addNewChild (
            Node *const child,
            Node *const parent,
            unsigned *const input )
```

Attachs a new child to a node.

Parameters

in	*child*	The child to attach
in,out	*parent*	The parent of the child.
in	*input*	The input sequence.

The input sequence is used to retrieve the key value of the hash table.

Definition at line 225 of file suffixTree.c.

4.8.3.2 addSuffixIndex()

```
void addSuffixIndex (
            Node *const node,
            const int labelLen,
            const size_t inputLen )
```

Add the suffix indexes to the leaves of the suffix tree.

Parameters

in,out	*node*	The root of the current subtree of the recursion.
in	*labelLen*	Length of the current suffix under consideration.
in	*inputLen*	Length of the input sequence.

Depth-first search traversal of the suffix tree to set the suffix indexes of the leaves. The one of the internal nodes are set to -1. The function traverses the tree in a recursive way adding the suffix indixes to the leaves.

Definition at line 75 of file suffixTree.c.

4.8.3.3 applyExtensions()

```
void applyExtensions (
            unsigned *const input,
            int *const remainder,
            ActivePoint *const ap )
```

Apply the extensions to the tree.

Parameters

in	*input*	The input sequence.
in,out	*remainder*	Counts the number of suffixes to be added.
in,out	*ap*	Current active point.

This function is applied once for each input byte and it performs several operation in loop until remainder is 0:

- If there is not an edge from the current active node such that the label on the edge starts with the same character of the one in input, it creates a new leaf starting from the current node.
- Otherwise it walks down the tree until either:
  - The current char being processed is already on the edge, in that case the function returns. The remainder is not zero so the algorithm can keep track of the suffix the are missing in the tree. They will be generated in the next phases.

– Otherwise an internal node is created because the input character is found in the middle of an edge.

Definition at line 108 of file suffixTree.c.

4.8.3.4 buildSuffixTree()

```
Node* buildSuffixTree (
            unsigned *const input,
            const size_t inputLen )
```

Returns the root of the suffix tree of the input sequence of bytes.

Note

The input sequence is assumed to terminate with a sentinel character, that is the only one present in the input.

Parameters

in	*input*	The input sequence from where the tree is built.
in	*inputLen*	The length of the input.

Returns

The root of the syntax tree of the sequence in input.

The function firstly initializes the active point and the root of the tree.
After that it performs n extensions of the suffix tree, where n is the length of the input string. Finally it adds the suffix indexes to the leaves of the tree and returns the root.
Basically the main function implementing the Ukkonen's algorithm is applyExtensions and is called a number of times equal to the length of the input string. Each time a new byte is processed.

Definition at line 42 of file suffixTree.c.

4.8.3.5 createInternalNode()

```
Node* createInternalNode (
            ActivePoint *const ap,
            HashChildren **const child,
            unsigned *const input )
```

Splits an edge by creating a new internal and a new leaf.

Parameters

in	*ap*	The current active point.
in,out	*child*	The child connected to the edge to be split.
in	*input*	The input sequence.

Returns

The internal node created.

The functions needs to update/insert various part of the tree around the edge that must be split and the. This is done in different steps:

- Delete the old child from the children of the current node.
- Create internal node
- Attach the old child as new child of the new internal node, then update the old child label, so that the new start of the node is the actual plus the active length.
- Create new leaf from the internal node.

Definition at line 182 of file suffixTree.c.

4.8.3.6 createLeaf()

```
void createLeaf (
            ActivePoint *const ap,
            Node *const parent,
            unsigned *const input )
```

Creates a leaf of the tree and attach it to the parent passed as arg.

Parameters

in	*ap*	The current acrive point.
in,out	*parent*	The parent of the new leaf.
in	*input*	The input sequence.

Definition at line 214 of file suffixTree.c.

4.8.3.7 createNode()

```
Node* createNode (
            const int start,
            int *const end,
            Node *const root )
```

Creates a node with default initialization.

Parameters

in	*start*	The start of the new node.
in	*end*	Pointer to the end of the new node.
in	*root*	The root of the suffix tree.

Returns

The new node created.

The default initialization of the node is performed by setting:

- The end, start passed as arguments.
- The root as suffix link.
- The set of children is NULL.
- The suffix index is set to -1.

Definition at line 165 of file suffixTree.c.

4.8.3.8 deleteChildren()

```
void deleteChildren (
            HashChildren ** children )
```

Frees the hash table that contains the children of a node.

Parameters

in	*children*	The hash table to be freed.

Definition at line 297 of file suffixTree.c.

4.8.3.9 deleteNode()

```
void deleteNode (
            Node * node )
```

Delete a node by free the memory for it and all of its children.

Parameters

in	*node*	The node to be deleted.

Definition at line 288 of file suffixTree.c.

4.8.3.10 getEdgeLen()

```
int getEdgeLen (
            Node *const node )
```

Returns the length of the label on the edge terminating in the input node.

Parameters

in	*node*	Node where the edge terminates.

Returns

The length of the label.

Definition at line 282 of file suffixTree.c.

4.8.3.11 printTree()

```
void printTree (
            Node * node,
            unsigned * input )
```

Depth-first traversal of the tree to print it.

Parameters

in	*node*	The root of the current subtree to be printed out.
in	*input*	The sequence in put.

Definition at line 307 of file suffixTree.c.

4.8.3.12 setSuffixLink()

```
void setSuffixLink (
            Node *const node,
            Node ** newest )
```

Sets a suffix link to a node.

Parameters

in	*node*	The node at which the suffix link will point.
in,out	*newest*	Pointer to the node which suffix link must be set.

Update suffix link if newest is pointing to a node, then reset newest for next iterations.
If newest is not null means that in the last iteration an internal node was created and now we set its suffix link.
The only node with suffixLink null is the root and all the other nodes have the root as default suffixLink.

Definition at line 243 of file suffixTree.c.

4.8.3.13 updateAP()

```
void updateAP (
            ActivePoint *const ap,
            const int remainder )
```

Update the active point.

Parameters

in,out	*ap*	The current pointer to the active point to update.
in	*remainder*	The remainder of the suffixes to be added.

Update the active point depending on whether the active node is root or not.

Definition at line 252 of file suffixTree.c.

4.8.3.14 walkDown()

```
int walkDown (
            ActivePoint *const ap,
            Node *const currentNode )
```

Walk down the tree onw level following one edge.

Parameters

in,out	*ap*	The current pointer to the acrive point.
in	*currentNode*	The current node from where to start walking.

Returns

1 if it has performed walkdown, 0 otherwise.

This function allows to walk down the tree quickly. In particulare if the active lenght is greater then the label length of the selected node, it can directly skip to that node by updating the active point.

Definition at line 268 of file suffixTree.c.

4.9 src/headers/unarith.h File Reference

Header file implementing an arithmetic coder compressor.

```
#include "arith.h"
```

Data Structures

- struct Decoder

Typedefs

- typedef struct Decoder Decoder

Functions

- Text arithDecoding (const Text input)

  Decode a sequence of bytes compressed by an arithmetic coder.
- void initDecoder (Decoder *const de, ByteBuffer *const inBuf, IOBuffer *const in, size_t inputLen)

  Initializes the decoder.
- unsigned decodeSymbol (Decoder *const de, Model *const model, ByteBuffer *const inBuf, IOBuffer *const in, size_t inputLen)

  Decode a sequence of bits and retrieve the symbol associated with the current interval read from the input sequence, that represents the interval in the model of probabilities.
- unsigned findChar (Decoder *const de, Interval *const interval, Model *const model)

  Selects the corresponding encoded symbol given a sequence of bits read. Sets also the interval of the decoded symbol and updates the model.
- void updateInterval (Decoder *const de, ByteBuffer *const inBuf, IOBuffer *const in, size_t inputLen)

  Updates the probability model.
- unsigned inputBit (Decoder *const de, ByteBuffer *const inBuf, IOBuffer *const in, size_t inputLen)

  Reads compressed bit and returns it.

4.9.1 Detailed Description

Header file implementing an arithmetic coder compressor.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the function needed to perform an airthmetic decompression. The code has been taken from https://github.com/wonder-mice/arcd even is it is slightly modified to make it fit better in the application.
The header offers the possibility to decode a given sequence of bytes in input, by using an adaptive model of probabilities, that must be the same of the one used to compress.
The parameters for the compression such as the number of bits to represent probability intervals are fixed because the purpose of the application was not to found the best compression possible, rather it aims to have an optimal compression time with a good compression rate.

4.9.2 Typedef Documentation

4.9.2.1 Decoder

```
typedef struct Decoder Decoder
```

Maintains the state of the decoder.

4.9.3 Function Documentation

4.9.3.1 arithDecoding()

```
Text arithDecoding (
            const Text input )
```

Decode a sequence of bytes compressed by an arithmetic coder.

Parameters

in	*input*	The sequence of bytes in input.

Returns

The sequence of bytes decoded.

This functions decompresses an input sequence of bytes using an arithmetic decoder and an adaptive model of probabilities.
It reads the sequence byte by byte and for each byte:

- Dcode the symbol using the actual probabilities..
- Update probabilities.

The probability model must be the same used for compressing the sequence.

Definition at line 60 of file unarith.c.

4.9.3.2 decodeSymbol()

```
unsigned decodeSymbol (
            Decoder *const de,
            Model *const model,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Decode a sequence of bits and retrieve the symbol associated with the current interval read from the input sequence, that represents the interval in the model of probabilities.

Parameters

in,out	*de*	Pointer to the decoder.
in,out	*model*	Pointer to the model of probabilities.
in,out	*inBuf*	Pointer to the buffer that keeps the current sequence of bits to decode.
in	*in*	Pointer to the sequence of bytes to decompress.
in	*inputLen*	The length of the sequence to decompress.

The operation is done by following three steps:

- Find char associated with the current state of the decoder and set the current interval of the symbol decoded.
- Update the current interval of the decoder with the read one.
- Update the probability model and check for other bits to read.

Definition at line 137 of file unarith.c.

4.9.3.3 findChar()

```
unsigned findChar (
            Decoder *const de,
            Interval *const interval,
            Model *const model )
```

Selects the corresponding encoded symbol given a sequence of bits read. Sets also the interval of the decoded symbol and updates the model.

Parameters

in	*de*	Pointer to the decoder.
out	*interval*	Pointer to the interval to be set.
in,out	*model*	Pointer to the model of probabilities.

Definition at line 160 of file unarith.c.

4.9.3.4 initDecoder()

```
void initDecoder (
            Decoder *const de,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Initializes the decoder.

Parameters

in,out	*de*	Pointer to the decoder to initialize.
in,out	*inBuf*	Pointer to the buffer that keeps the current sequence of bits to decode.
in	*in*	Pointer to the sequence of bytes to decompress.
in	*inputLen*	The length of the sequence to decompress.

The decoder is initialize so that the intervals is the total interval [0,1] and the read values are zero. The first 17 bits are read from the input.

Definition at line 119 of file unarith.c.

4.9.3.5 inputBit()

```
unsigned inputBit (
            Decoder *const de,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Reads compressed bit and returns it.

Parameters

in	*de*	Pointer to the decoder.
in,out	*inBuf*	Pointer to the buffer that keeps the current sequence of bits to decode.
in	*in*	Pointer to the sequence of bytes to decompress.
in	*inputLen*	The length of the sequence to decompress.

Returns

This function reads one compressed bit from the input buffer and returns it. When the end of file is reached, byt the fin variable of the decoder is set to 1, this functions returns the continuation bit to let the decoder finish the decompression.

Definition at line 222 of file unarith.c.

4.9.3.6 updateInterval()

```
void updateInterval (
            Decoder *const de,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Updates the probability model.

Parameters

in	*de*	Pointer to the decoder.
in,out	*inBuf*	Pointer to the buffer that keeps the current sequence of bits to decode.
in	*in*	Pointer to the sequence of bytes to decompress.
in	*inputLen*	The length of the sequence to decompress.

Definition at line 183 of file unarith.c.

4.10 src/headers/unbwt.h File Reference

Header file implementing the reverse Burrows-Wheeler transformation.

```
#include "util.h"
```

Functions

- Text unbwt (Text input)

  Returns the reverse BWT of the input sequence of bytes.

4.10.1 Detailed Description

Header file implementing the reverse Burrows-Wheeler transformation.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

The unique function offered by the interface allows to reverse the BWT of the string.
The input string must have the index of the first char of the original string, as well as the index of the sentinel char encoded in 4 bytes each at the beginning of the input.

4.10.2 Function Documentation

4.10.2.1 unbwt()

```
Text unbwt (
            Text input )
```

Returns the reverse BWT of the input sequence of bytes.

Parameters

in	*input*	The sequence of bytes to be reversed.

Returns

The reverse BWT of the input string.

The implementation follows the one based on the LF mapping, a function from the last column of the BWT to the first column. It is based on the lemma that the i-th occurrence of char c in the last column of the BWT matrix is the same exact character as the i-th occurrence c in the first column.
The algorithm starts reading the index of the first and of the sentinel characters. Then it calculates the frequencies of each character in the input sequence. After that it computes two arrays:

- One array that contains for each possible value i the number of values that are alphabetically smaller than i (mappings).
- One array that contains the LF mapping (links) calculated from mappings and frequencies.

Finally it outputs the original string from the LF mapping.

Definition at line 50 of file unbwt.c.

4.11 src/headers/unmtf.h File Reference

Header file implementing the reverse of the move to front transformation.

```
#include "mtf.h"
```

Functions

- Text unmtf (const Text input)
  Reverses the move to front transformation of the input sequence of bytes.
- unsigned char searchSymbol (SymbolsList *const symbols, const unsigned char pos, SymbolsList **aux)
  Search the symbol in the list at the position specified in input.

4.11.1 Detailed Description

Header file implementing the reverse of the move to front transformation.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the functions needed to reverse the move to front transformation. The main function is unmtf that takes as input a sequence of bytes and reverses the transformation returning the original sequence of bytes. The utility function searchSymbol search the symbol in input in the list of symbols having as input the position of the symbol.

4.11.2 Function Documentation

4.11.2.1 searchSymbol()

```
unsigned char searchSymbol (
            SymbolsList *const symbols,
            const unsigned char pos,
            SymbolsList ** aux )
```

Search the symbol in the list at the position specified in input.

Parameters

in	*symbols*	Pointer to the head of the list of symbols.
in	*pos*	Position of the symbol to be searched.
in,out	*aux*	Pointer to the previous element of the searched symbol in the list, useful later to move ahead the returned element.

Returns

The symbol in the list of symbols at the position specified.

The function implements a simple search of an element in a list of bytes. It outputs the corresponding element at the specified position and, at the same time, it stores in the double pointer aux the element before the one searched in the list, so as to make the next operation of moving ahead the element quicker.

Definition at line 72 of file unmtf.c.

4.11.2.2 unmtf()

```
Text unmtf (
            const Text input )
```

Reverses the move to front transformation of the input sequence of bytes.

Parameters

in	*input*	The input sequence to be reversed.

Returns

The reversed MTF of the input.

The function is the symmetrical of the MTF transformation. In fact, it searches in the list of symbols the symbol which position is the one in the input, then it outputs the symbol and updates the list, moving the element in front of that. At the beginning, the symbols list is initialized as for the forward transformation.

Definition at line 40 of file unmtf.c.

4.12 src/headers/unzle.h File Reference

Header file implementing the reverse zero-length encoding.

```
#include "util.h"
```

Functions

- Text zleDecoding (const Text input)

  Reverse the zero-length encoding routine.
- void outputZeroes (unsigned char *runLen, unsigned char *out, size_t *index)

  Writes in the output array the sequences of zeroes encoded by the value pointed by runLen.

4.12.1 Detailed Description

Header file implementing the reverse zero-length encoding.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This interface defines the functions needed to reverse the zero-length encoding routine of the BWT compression. In particular, the function zleDecoding takes as input a sequence of bytes and undos the ZLE. Another function outputZeroes is used to reverse the binary number that represents the number of zeroes of the original string.
For the format of the input string refers to the zero-length encoding interfaces zle.h

4.12.2 Function Documentation

4.12.2.1 outputZeroes()

```
void outputZeroes (
            unsigned char * runLen,
            unsigned char * out,
            size_t * index )
```

Writes in the output array the sequences of zeroes encoded by the value pointed by runLen.

Parameters

in	*runLen*	Pointer to the binary number representing the number of zeroes of the original input.
in,out	*out*	Pointer to the output array where writes the zeroes.
in,out	*index*	Pointer to the index of the next cell to be written in the output array.

Definition at line 92 of file unzle.c.

4.12.2.2 zleDecoding()

```
Text zleDecoding (
            const Text input )
```

Reverse the zero-length encoding routine.

Parameters

in	*input*	The input sequence of bytes to be rebersed.

Returns

The reverse ZLE of the input.

Definition at line 33 of file unzle.c.

4.13 src/headers/util.h File Reference

Header file implementing a util interface for encoding bytes and managing files.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <sys/time.h>
```

Data Structures

- struct Text

Macros

- #define POW_2(N) (1 << (N))
  2^N.
- #define MAX_CHUNK_SIZE (5 * 1024 * 1024)
  Max chunk size allowed.
- #define DEFAULT_CHUNK_SIZE (0.9 * 1024 * 1024)
  Default chunk size.
- #define MIN_CHUNK_SIZE (300)
  Min size for a chunk to be compressed.

Typedefs

- typedef struct Text Text

Functions

- unsigned char decomposeUnsigned (unsigned u, int n)

  Utility function to encode an unsigned.
- void encodeUnsigned (const unsigned u, unsigned char ∗const output, int outIndex)

  Stores in an array of unsigned char that represents ul big-endian.
- unsigned readUnsigned (unsigned char ∗const input, size_t n)

  Returns the unsigned encoded big endian by the 4 bytes of the input from location n.
- FILE ∗ openFileRB (char ∗const filename)

  Open a file for reading it.
- Text readFile (FILE ∗const file, long size)

  Reads at most size bytes from the file in input.
- FILE ∗ openFileWB (char ∗const filename)

  Opens a file for wiriting into it.
- void writeFile (FILE ∗const file, unsigned char ∗const buffer, long len)

  Writes the buffer into the file.
- long fileSize (FILE ∗const file)

  gets the number of bytes of the file.
- int compareFiles (FILE ∗const file1, FILE ∗const file2, long size1, long size2)

  Compare two files byte by byte.

4.13.1 Detailed Description

Header file implementing a util interface for encoding bytes and managing files.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This interface defines some functions for encoding unsigned values and for managinfg files.

4.13.2 Typedef Documentation

4.13.2.1 Text

```
typedef struct Text Text
```

This structure is the one who stores the sequence of bytes to be compress or decompress together with its length. In addition, the id variable stores the id of the chunk during compression. This is foundamental for the parallel version of the compression algorithm, since the execution of the chunk compression could be out of order, so they need to be ordered at the end of the compression.

4.13.3 Function Documentation

4.13.3.1 compareFiles()

```
int compareFiles (
            FILE *const file1,
            FILE *const file2,
            long size1,
            long size2 )
```

Compare two files byte by byte.

Parameters

in	*file1*	The first file to compare.
in	*file2*	The second file to compare.
in	*size1*	The number of bytes of the first file-
in	*size2*	The number of bytes of the second files.

Returns

- 2 if the two sizes are different.
- 1 if the two files are equal.
- 0 if the two files are different.

Definition at line 150 of file util.c.

4.13.3.2 decomposeUnsigned()

```
unsigned char decomposeUnsigned (
            unsigned u,
            int n )
```

Utility function to encode an unsigned.

Parameters

in	*u*	The unsigned to decompose.
in	*n*	The byte of the decomposed unsigned.

Returns

Returns the nth big-endian byte of an unsigned int. (0 <= n <= 3)

Definition at line 34 of file util.c.

4.13.3.3 encodeUnsigned()

```
void encodeUnsigned (
            const unsigned u,
            unsigned char *const output,
            int outIndex )
```

Stores in an array of unsigned char that represents ul big-endian.

Parameters

in	*u*	The unsigned to encode.
out	*output*	Pointer to the output unsigned char where the result is stored.
in,out	*outIndex*	Index of the first element of the output array to be write.

Definition at line 39 of file util.c.

4.13.3.4 fileSize()

```
long fileSize (
            FILE *const file )
```

gets the number of bytes of the file.

Parameters

in	*file*	Input file.

Returns

The number of bytes of the file.

Definition at line 139 of file util.c.

4.13.3.5 openFileRB()

```
FILE* openFileRB (
            char *const filename )
```

Open a file for reading it.

Parameters

in	*filename*	The filename of the file to be opened.

Returns

The file construct of the opened file.

Definition at line 60 of file util.c.

4.13.3.6 openFileWB()

```
FILE* openFileWB (
            char *const filename )
```

Opens a file for wiriting into it.

Parameters

in	*filename*	The filename of the file to be opened.

Returns

The file construct of the opened file.

Definition at line 95 of file util.c.

4.13.3.7 readFile()

```
Text readFile (
            FILE *const file,
            long size )
```

Reads at most size bytes from the file in input.

Parameters

in	*file*	The file to be read.
in	*size*	The number of bytes to read.

Returns

The sequence of bytes of the size specified read in the file in input.

Definition at line 75 of file util.c.

4.13.3.8 readUnsigned()

```
unsigned readUnsigned (
            unsigned char *const input,
            size_t n )
```

Returns the unsigned encoded big endian by the 4 bytes of the input from location n.

Parameters

in	*input*	Pointer to the encoded unsigned int.
in	*n*	The index of the first byte of the encoded unsigned.

Returns

Returns the unsigned int encoded big-endian by the 4 bytes starting at location n.

Definition at line 47 of file util.c.

4.13.3.9 writeFile()

```
void writeFile (
            FILE *const file,
            unsigned char *const buffer,
            long len )
```

Writes the buffer into the file.

Parameters

in	*file*	The file to be written.
in	*buffer*	The buffer to be written into the file.
in	*len*	The length of the sequence of bytes to be written.

Definition at line 126 of file util.c.

4.14 src/headers/zle.h File Reference

Header file implementing a zero length encoder.

```
#include "util.h"
```

Functions

- Text zleEncoding (const Text input)

  Performs the zero length encoding.
- void encodeZeroRun (size_t runLen, Text *res)

  Encodes and stores the zero-run in input.

4.14.1 Detailed Description

Header file implementing a zero length encoder.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

This header defines the functions needed to perform the zero length encoding, third phase of the BWT compression.
This type of compression has been developed by D.J.Wheeler and it is a kind of run-length encoding, that encodes sequences of zeroes in a number that holds the length of the zero-run.
Given a sequence of zeroes of length N, it converts (N + 1) to binary and outputs the binary representation, starting from the LSB and leaving out the MSB.

Examples

- Cleartext sequence of zeroes of length: 1, 2, 3, 4, 5, 6, 7, 8, 9
- Encoded: N + 1 in binary, MSB elided: 0, 1, 00, 01, 10, 11, 000, 001, 010

- Cleartext: A byte B = 0x01 or 0x02 or ... or 0xFD
- Encoded: B + 1

- Cleartext: A 0xFE byte
- Encoded: 0xFF 0x00

- Cleartext: A 0xFF byte
- Encoded: 0xFF 0x01

See also

Burrows Wheeler Compression: Principles and Reflections, Peter Fenwick, Private Bag 92019, Auckland, New Zealand.

4.14.2 Function Documentation

4.14.2.1 encodeZeroRun()

```
void encodeZeroRun (
            size_t runLen,
            Text * res )
```

Encodes and stores the zero-run in input.

Parameters

in	*runLen*	The run of zeroes to encode.
in,out	*res*	Pointer to the actual result.

This function encodes a sequence of zeroes of length zeroRun in the big-endian representation of (zeroRun + 1), eliding the MSB.

Definition at line 92 of file zle.c.

4.14.2.2 zleEncoding()

```
Text zleEncoding (
            const Text input )
```

Performs the zero length encoding.

Parameters

in	*input*	Sequence of bytes together with its length.

Returns

The text encoded as describe above.

The main method of the file encodes a sequence of length N by compressing sequences of consecutive zeroes. Each one of that will be encoded in a binary number that represents the length of the zero-run. Each byte in input is encoded in an online fashion, written it down as soon as it is read, while each sequence of zeroes is encoded as soon as it ends by calling the function encodeZeroRun.

Definition at line 44 of file zle.c.

4.15 src/sources/arith.c File Reference

Source file implementing an arithmetic coder compressor.

```
#include "../headers/arith.h"
```

Macros

- #define BUF_BITS (8 ∗ sizeof(unsigned char))

  Number of bits of the byte buffer.

Functions

- Text encodingRoutine (const Text input)

  Main function of the arithmetic encoder.
- void initEncoder (Encoder ∗const encoder)

  Initialize the encoder pointed by the pointer in input.
- void encodeSymbol (unsigned ∗const low, unsigned ∗const high, const unsigned range, const Interval interval)

  Encode the symbol by scaling the range basing on the interval of the symbol.
- void findInterval (Interval ∗const interval, Model ∗const model, const unsigned ch)

  Find the interval probability of the input symbol.
- void finishEncoding (Encoder ∗const en, ByteBuffer ∗const byteBuf, IOBuffer ∗const buf)

  It checks if there are still bits to write in output, and writes it.
- void initModel (Model ∗const model, const unsigned size)

  Initialization of the mdoel. Assigns a uniform probability to all the values.
- void updateModel (Model ∗const model, const unsigned ch)

  Updates the model basing on the new occurrence of the character in input.
- void initByteBuffer (ByteBuffer ∗const byteBuf)

  Byte buffer initialization.
- void checkForOutputBit (Encoder ∗const encoder, ByteBuffer ∗const byteBuf, IOBuffer ∗const buf)

  Check if there are bits that can be output and it output those bit in the byte buffer.
- void outputBits (Encoder ∗const en, const unsigned bit, ByteBuffer ∗const byteBuf, IOBuffer ∗const buffer)

  Output the bit in input in the byte Buffer and checks and outputs underflow bits.
- void outputBit (ByteBuffer ∗const byteBuf, const unsigned bit, IOBuffer ∗const buffer)

  Writes the bit in the byte buffer and then, if the byte buffer is full, it empties it into the buffer.

4.15.1 Detailed Description

Source file implementing an arithmetic coder compressor.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.15.2 Function Documentation

4.15.2.1 checkForOutputBit()

```
void checkForOutputBit (
            Encoder *const encoder,
            ByteBuffer *const byteBuf,
            IOBuffer *const buf )
```

Check if there are bits that can be output and it output those bit in the byte buffer.

Parameters

in,out	*encoder*	The encoder.
in,out	*byteBuf*	The byte buffer.
in,out	*buf*	The buffer for output.

Definition at line 202 of file arith.c.

4.15.2.2 encodeSymbol()

```
void encodeSymbol (
            unsigned *const low,
            unsigned *const high,
            const unsigned range,
            const Interval interval )
```

Encode the symbol by scaling the range basing on the interval of the symbol.

Parameters

in,out	*low*	Current low value of the interval.
in,out	*high*	Current high value of the interval.
in	*range*	Current range.
in	*interval*	Interval relative to the current symbol in input.

Definition at line 118 of file arith.c.

4.15.2.3 encodingRoutine()

```
Text encodingRoutine (
            const Text input )
```

Main function of the arithmetic encoder.

This functions compresses an input sequence of bytes using an arithmetic encoder and an adaptive model of probabilities.
It reads the sequence byte by byte and for each byte:

- Encode the symbol using the actual probabilities.
- Check if the ByteBuffer is full to print out a char.
- Update probabilities.

Definition at line 51 of file arith.c.

4.15.2.4 findInterval()

```
void findInterval (
            Interval *const interval,
            Model *const model,
            const unsigned ch )
```

Find the interval probability of the input symbol.

Parameters

out	*interval*	The interval that needs to be found.
in	*model*	The model of probabilities.
in	*ch*	The character which interval has to be found.

Definition at line 128 of file arith.c.

4.15.2.5 finishEncoding()

```
void finishEncoding (
            Encoder *const encoder,
            ByteBuffer *const byteBuf,
            IOBuffer *const buf )
```

It checks if there are still bits to write in output, and writes it.

Parameters

in	*encoder*	The encoder.
in,out	*byteBuf*	The byte buffer.
in,out	*buf*	The buffer for output.

Definition at line 140 of file arith.c.

4.15.2.6 initByteBuffer()

```
void initByteBuffer (
            ByteBuffer * byteBuf )
```

Byte buffer initialization.

Parameters

in,out	*byteBuf*	

Definition at line 195 of file arith.c.

4.15.2.7 initEncoder()

```
void initEncoder (
            Encoder *const encoder )
```

Initialize the encoder pointed by the pointer in input.

The encoder is initialize so that the intervals is the total interval [0,1].

Definition at line 109 of file arith.c.

4.15.2.8 initModel()

```
void initModel (
            Model *const model,
            const unsigned size )
```

Initialization of the mdoel. Assigns a uniform probability to all the values.

Parameters

in,out	*model*	The model to initialize.
in	*size*	The size of the model, i.e. the number of probabilities.

Definition at line 164 of file arith.c.

4.15.2.9 outputBit()

```
void outputBit (
            ByteBuffer *const byteBuf,
            const unsigned bit,
            IOBuffer *const buffer )
```

Writes the bit in the byte buffer and then, if the byte buffer is full, it empties it into the buffer.

Parameters

in,out	*byteBuf*	The byte buffer.
in	*bit*	The bit to be printed.
in,out	*buffer*	The buffer.

Definition at line 245 of file arith.c.

4.15.2.10 outputBits()

```
void outputBits (
            Encoder *const encoder,
            const unsigned bit,
            ByteBuffer *const byteBuf,
            IOBuffer *const buffer )
```

Output the bit in input in the byte Buffer and checks and outputs underflow bits.

Parameters

in,out	*encoder*	The encoder.
in	*bit*	The bit to be printed out.
in,out	*byteBuf*	The byte buffer to store the current byte to be printed.
in,out	*buffer*	The buffer to store the current overall output.

Definition at line 233 of file arith.c.

4.15.2.11 updateModel()

```
void updateModel (
            Model *const model,
            const unsigned ch )
```

Updates the model basing on the new occurrence of the character in input.

Parameters

in,out	*model*	The model to be updated.
in	*ch*	The new occurrence for which the model has to be updated.

Definition at line 174 of file arith.c.

4.16 src/sources/bwt.c File Reference

Sourcefile implementing the Burrows-Wheeler transformation (BWT).

```
#include "../headers/bwt.h"
#include "../extern/uthash.h"
```

Macros

- #define SENTINEL 256;

  Sentinel character.

Functions

- Text bwtTransformation (const Text input)

  Returns the BWT transformation of the input text.
- Text getBWT (unsigned ∗const input, const size_t inputLen, Node ∗const suffixTree)

  Returns the BWT transformation of the input syntax tree.
- void createSuffixArray (Node ∗const node, unsigned ∗const input, int ∗const i, int ∗const suffixArray)

  Creates the suffix array recursively, starting from the root of the suffix tree in input and frees the suffix tree.
- unsigned ∗ setSentinel (Text input)

  Set the sentinel at the end of the input Text returning a pointer to a sequence of unsigned.

4.16.1 Detailed Description

Sourcefile implementing the Burrows-Wheeler transformation (BWT).

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.16.2 Function Documentation

4.16.2.1 bwtTransformation()

```
Text bwtTransformation (
            const Text input )
```

Returns the BWT transformation of the input text.

This function return the BWT transformation of the input text in three phases:

- First set the sentinel character in the input text. This step is fundamental to obtain the suffix array, since the transformation is reversible if and only if the input text ends with a character different from all the others.
- Second the suffix tree is built.
- Finally the BWT is created by calling the function getBWT.

Definition at line 44 of file bwt.c.

4.16.2.2 createSuffixArray()

```
void createSuffixArray (
            Node *const node,
            unsigned *const input,
            int *const i,
            int *const suffixArray )
```

Creates the suffix array recursively, starting from the root of the suffix tree in input and frees the suffix tree.

Lexicographic depth-first search traversal to create the suffix array. The suffix tree is recursively traversed and as soon as a node and all its children have been visited they are deleted.

Definition at line 116 of file bwt.c.

4.16.2.3 getBWT()

```
Text getBWT (
            unsigned *const input,
            const size_t inputLen,
            Node *const suffixTree )
```

Returns the BWT transformation of the input syntax tree.

This function encodes the BWT in the output array of bytes. To make the transformation reversible we store in the output array also the index of the sentinel character and the index of the first character of the input. These two index can be bigger than a byte, so they are encoded in two bytes each.
The output array will have:

- First two bytes storing the encode unsigned index of the first char.
- Third and fourth byte storing the encode usigned index of the sentinel.
- The sentinle character itself is encoded as zero.

In this way we can maintain the result in an array of unsigned char.
The operation is performed in two steps:

- Create suffix array starting from the suffix tree.
- Extract the BWT from the suffix array.

Definition at line 73 of file bwt.c.

4.16.2.4 setSentinel()

```
unsigned* setSentinel (
             Text input )
```

Set the sentinel at the end of the input Text returning a pointer to a sequence of unsigned.

This functions returns the input text with the sentinel character concatenated at the end. The pointer returned is unsigned so that it can contains also the sentinel character (i.e. char 256). Since the input text is of type unsigned char, we cna be sure that the sentinel is the only character in the text.

Definition at line 156 of file bwt.c.

4.17 src/sources/bwtUnzip.c File Reference

Source file implementing the sequential version of the Burrows-Wheeler decompression.

```
#include "../headers/bwtUnzip.h"
```

Functions

- Text bwtUnzip (const Text input)

  Performs the BWT decompression.
- void decompress (FILE *input, FILE *output)

  Decompress a given input file, writing the result in the specified output file.

4.17.1 Detailed Description

Source file implementing the sequential version of the Burrows-Wheeler decompression.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.17.2 Function Documentation

4.17.2.1 bwtUnzip()

```
Text bwtUnzip (
            const Text input )
```

Performs the BWT decompression.

This function applies in reverse the four step of the BWT decompression in sequence to a single sequence of bytes. The main phase are:

1. Arithmetic decoding.
2. Zero-length deconding.
3. MTF reverse transformation.
4. BWT reverse transformation.

Definition at line 38 of file bwtUnzip.c.

4.17.2.2 decompress()

```
void decompress (
            FILE * input,
            FILE * output )
```

Decompress a given input file, writing the result in the specified output file.

The compressed file is read. First the encoded length and the id are extract from the file. Then if the id is 1 the compressed chunk of the length read before is decompressed and written out in the output file. The code is consistency with the compression version.

Definition at line 61 of file bwtUnzip.c.

4.18 src/sources/mtf.c File Reference

Source file implementing the move to front transformation.

```
#include "../headers/mtf.h"
```

Macros

- #define SIZE_SYMBOLS_LIST 256

  Size of the list of symbols.

Functions

- Text mtf (const Text input)

  Transform the input sequence of bytes into an array of output bytes, indices of the symbol list.
- int search (SymbolsList *const symbols, const unsigned char byte, SymbolsList **aux)

  Returns the position of the searched byte in the list of symbols, or -1 if the element is not found.
- SymbolsList * mvtElement (SymbolsList *const symbols, SymbolsList *el)

  Move the element pointed by el to the front of the list in input.
- SymbolsList * initListOfSymbols ()
- void freeListOfSymbols (SymbolsList *symbols)

4.18.1 Detailed Description

Source file implementing the move to front transformation.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.18.2 Function Documentation

4.18.2.1 freeListOfSymbols()

```
void freeListOfSymbols (
            SymbolsList * symbols )
```

Free the list of symbols and deallocate memory.

Definition at line 152 of file mtf.c.

4.18.2.2 initListOfSymbols()

```
SymbolsList* initListOfSymbols (
            void  )
```

Populate the list of symbols with characters from 0 to 255

Definition at line 129 of file mtf.c.

4.18.2.3 mtf()

```
Text mtf (
            const Text input )
```

Transform the input sequence of bytes into an array of output bytes, indices of the symbol list.

The function scan the input array of bytes and for each byte it performs three actions:

- Search the position of the input byte.
- Write the position in the output array.
- Move to the head of the symbol list the read byte.

Definition at line 37 of file mtf.c.

4.18.2.4 mvtElement()

```
SymbolsList* mvtElement (
            SymbolsList *const symbols,
            SymbolsList * el )
```

Move the element pointed by el to the front of the list in input.

This function moves the element pointed by the one given in input to the front of the list of symbols. The function is carried out with two operations:

- Delete element from the list of symbols.
- Re-insert it at the head of the list.

Definition at line 112 of file mtf.c.

4.18.2.5 search()

```
int search (
            SymbolsList *const symbols,
            const unsigned char byte,
            SymbolsList ** aux )
```

Returns the position of the searched byte in the list of symbols, or -1 if the element is not found.

The function implements a simple search of an element in a list of bytes. It outputs the corresponding position of the element and, at the same time, it stores in the double pointer aux the element before the one searched in the list, so as to make the next operation of moving ahead the element quicker.

Definition at line 79 of file mtf.c.

4.19 src/sources/parallelBwtZip.c File Reference

Source file implementing the parallel version of the Burrows-Wheeler compression.

```
#include "../headers/parallelBwtZip.h"
```

Functions

- void compressParallel (FILE *input, FILE *output, const long chunkSize)
  Performs the parallel BWT compression.
- void * bwtStage (void *arg)
  Thread for the Burrow-Wheeler transform stage.
- void * mtfZleStage (void *arg)
  Thread for the MTF and ZLE stage.
- void * arithStage (void *arg)
  Thread for the arithmetic coding stage.
- void initBuffer (Buffer *const buf)
  Initialize a queue buffer.
- void writeOutput (FILE *output, int *const index, const int littleChunk)
  Write the compressed output from the result array to the output file.
- void setAffinity (cpu_set_t *const cpus, int cpu, pthread_attr_t *const attr)
  Set a specific core where a thread will be run.

Variables

- Buffer readin = {NULL, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER}
- Buffer bwt = {NULL, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER}
- Buffer arith = {NULL, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER}
- Text * result
- int nBlocks

4.19.1 Detailed Description

Source file implementing the parallel version of the Burrows-Wheeler compression.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.19.2 Function Documentation

4.19.2.1 arithStage()

```
void* arithStage (
            void * arg )
```

Thread for the arithmetic coding stage.

Performs the bwt on a chunk of data arith from the read buffer as soon it is available. Note that the thread must first lock the mutex on the arith buffer and checks that it is not empty. Then it can read the data and perform the operation. Finally it needs to lock the lock on the output array, in this case the result array, before written the result into the array.

Definition at line 285 of file parallelBwtZip.c.

4.19.2.2 bwtStage()

```
void* bwtStage (
            void * arg )
```

Thread for the Burrow-Wheeler transform stage.

Performs the bwt on a chunk of data read from the read buffer as soon it is available. Note that the thread must first lock the mutex on the read buffer and checks that it is not empty. Then it can read the data and perform the operation. Finally it needs to lock the lock on the output buffer, in this case the bwt buffer, before written the result into the buffer.

Definition at line 202 of file parallelBwtZip.c.

4.19.2.3 compressParallel()

```
void compressParallel (
            FILE * input,
            FILE * output,
            const long chunkSize )
```

Performs the parallel BWT compression.

This function applies the BWT compression to a file in input. The main thread creates other 8 threads: 6 for the BWT phase, 1 for the MTF and ZLE phases and one for the arithmetic phase. Threads communicate with each other by using shared global buffers. Each thread, including the main one, acts as consumer as well as a producer. In fact:

- The main thread is the producer of the read buffer because it fills in it after the read of one chunk of bytes from the input file. Then it acts as a consumer of the result array, because it has to write the compressed chunk store in the result in the output file.
- The bwt thread is the consumer of the read buffer and the producer of the bwt buffer.
- The mtf and zle thread is the consumer of the bwt buffer and the producer of the arith buffer.
- Finally the arith thread is the consumer of the arith buffer and the producer of the result array.

Each chunk of data is processed asynchronously and they are written in order given the id of the chunk, setted before the compression starts.

Definition at line 91 of file parallelBwtZip.c.

4.19.2.4 initBuffer()

```
void initBuffer (
            Buffer *const buf )
```

Initialize a queue buffer.

Parameters

in,out	*buf*	The buffer to be initialized.

Definition at line 316 of file parallelBwtZip.c.

4.19.2.5 mtfZleStage()

```
void* mtfZleStage (
            void * arg )
```

Thread for the MTF and ZLE stage.

Performs the bwt on a chunk of data read from the bwt buffer as soon it is available. Note that the thread must first lock the mutex on the bwt buffer and checks that it is not empty. Then it can read the data and perform the operation. Finally it needs to lock the lock on the output buffer, in this case the arith buffer, before written the result into the buffer.

Definition at line 243 of file parallelBwtZip.c.

4.19.2.6 setAffinity()

```
void setAffinity (
            cpu_set_t *const cpus,
            int cpu,
            pthread_attr_t *const attr )
```

Set a specific core where a thread will be run.

Parameters

in	*cpus*	Pointer to the number of cpus.
in,out	*cpu*	The cpu number to be set for the thread.
in,out	*attr*	Pointer to the attribute of the thread that must run on the specific core.

Definition at line 356 of file parallelBwtZip.c.

4.19.2.7 writeOutput()

```
void writeOutput (
            FILE * output,
            int *const index,
            const int littleChunk )
```

Write the compressed output from the result array to the output file.

The main thread continuously calls this function to check if the next block to be output is available.
The index keeps trace of the id of the next block to output. The id of the result are initialized to -1 so the main thread knows when a compressed block has been stored into the array.

Definition at line 329 of file parallelBwtZip.c.

4.19.3 Variable Documentation

4.19.3.1 arith

```
Buffer arith = {NULL, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER}
```

Shared buffer to hold the intermediate result between the ZLE phase and the arithmetic coding phase.

Definition at line 49 of file parallelBwtZip.c.

4.19.3.2 bwt

```
Buffer bwt = {NULL, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER}
```

Shared buffer to hold the intermediate result between the BWT stage and the MTF stage.

Definition at line 43 of file parallelBwtZip.c.

4.19.3.3 nBlocks

```
int nBlocks
```

Number of blocks (chunks) of the input file to be compressed.

Definition at line 60 of file parallelBwtZip.c.

4.19.3.4 readin

```
Buffer readin = {NULL, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER}
```

Shared buffer where the chunks are stored before they begin the compression.

Definition at line 37 of file parallelBwtZip.c.

4.19.3.5 result

```
Text* result
```

Output array to hold the compressed chunks in order before they are written into the output file.

Definition at line 55 of file parallelBwtZip.c.

4.20 src/sources/queue.c File Reference

Source file implementing a queue.

```
#include "../headers/queue.h"
```

Functions

- void initQueue (Queue *q)
  Queue initialization.
- void enqueue (Text elem, Queue *q)
  Adds an element at the front of the queue.
- Text dequeue (Queue *q)
  Deletes an element from the rear of the queue and returns it.
- int empty (Queue *q)
  Checks if the queue is empty.

4.20.1 Detailed Description

Source file implementing a queue.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.20.2 Function Documentation

4.20.2.1 dequeue()

```
Text dequeue (
            Queue * q )
```

Deletes an element from the rear of the queue and returns it.

Parameters

in,out	*q*	The queue where the element must be retrieved.

Returns

The element dequeued.

Definition at line 58 of file queue.c.

4.20.2.2 empty()

```
int empty (
            Queue * q )
```

Checks if the queue is empty.

Parameters

in	*q*	The queue.

Returns

- 1 if the queue is empy.
- 0 otherwise.

Definition at line 74 of file queue.c.

4.20.2.3 enqueue()

```
void enqueue (
            Text elem,
            Queue * q )
```

Adds an element at the front of the queue.

Parameters

in	*elem*	The element to be added.
in,out	*q*	The queue qhere the element must be added.

Definition at line 41 of file queue.c.

4.20.2.4 initQueue()

```
void initQueue (
            Queue * q )
```

Queue initialization.

Parameters

in,out	*q*	The queue to initialize.

Definition at line 33 of file queue.c.

4.21 src/sources/sequentialBwtZip.c File Reference

Source file implementing the sequential version of the Burrows-Wheeler compression.

```
#include "../headers/sequentialBwtZip.h"
```

Functions

- Text bwtZip (const Text input)
  Performs the sequential BWT compression.
- void compressSequential (FILE *input, FILE *output, long chunkSize)
  Compress a given input file, writing the result in the specified output file.

4.21.1 Detailed Description

Source file implementing the sequential version of the Burrows-Wheeler compression.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.21.2 Function Documentation

4.21.2.1 bwtZip()

```
Text bwtZip (
            const Text input )
```

Performs the sequential BWT compression.

This function applies the four step of the BWT compression in sequence to a single sequence of bytes.
The main phase are:

1. BWT transformation.
2. MTF transformation.
3. Zero-length encoding.
4. Arithmetic compression.

Definition at line 38 of file sequentialBwtZip.c.

4.21.2.2 compressSequential()

```
void compressSequential (
            FILE * input,
            FILE * output,
            long chunkSize )
```

Compress a given input file, writing the result in the specified output file.

The file is read chunk by chunk until is finished. Each chunk is compressed when is read and then written in output folloring its length (encoded in 4 bytes) and the id of the chunk. If one chunk is smaller than MIN_CHUNK_SIZE the chunk is not compressed for efficiency reasons.
The id is 1 if the chunk is compressed, 0 otherwise.

Definition at line 63 of file sequentialBwtZip.c.

4.22 src/sources/suffixTree.c File Reference

Source file implementing the construction of the suffix tree given a sequence of bytes.

```
#include "../headers/suffixTree.h"
```

Functions

- Node * buildSuffixTree (unsigned *const input, const size_t inputLen)

  Returns the root of the suffix tree of the input sequence of bytes.
- void addSuffixIndex (Node *const node, const int labelLen, const size_t inputLen)

  Add the suffix indexes to the leaves of the suffix tree.
- void applyExtensions (unsigned *const input, int *const remainder, ActivePoint *const ap)

  Apply the extensions to the tree.
- Node * createNode (const int start, int *const end, Node *const root)

  Creates a node with default initialization.
- Node * createInternalNode (ActivePoint *const ap, HashChildren **const child, unsigned *const input)

  Splits an edge by creating a new internal and a new leaf.
- void createLeaf (ActivePoint *const ap, Node *const parent, unsigned *const input)

  Creates a leaf of the tree and attach it to the parent passed as arg.
- void addNewChild (Node *const child, Node *const parent, unsigned *const input)

  Attachs a new child to a node.
- void setSuffixLink (Node *const node, Node **newest)

  Sets a suffix link to a node.
- void updateAP (ActivePoint *const ap, const int remainder)

  Update the active point.
- int walkDown (ActivePoint *const ap, Node *const currentNode)

  Walk down the tree onw level following one edge.
- int getEdgeLen (Node *const node)

  Returns the length of the label on the edge terminating in the input node.
- void deleteNode (Node *node)

  Delete a node by free the memory for it and all of its children.
- void deleteChildren (HashChildren **children)

  Frees the hash table that contains the children of a node.
- void printTree (Node *node, unsigned *input)

  Depth-first traversal of the tree to print it.

4.22.1 Detailed Description

Source file implementing the construction of the suffix tree given a sequence of bytes.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.22.2 Function Documentation

4.22.2.1 addNewChild()

```
void addNewChild (
            Node *const child,
            Node *const parent,
            unsigned *const input )
```

Attachs a new child to a node.

The input sequence is used to retrieve the key value of the hash table.

Definition at line 225 of file suffixTree.c.

4.22.2.2 addSuffixIndex()

```
void addSuffixIndex (
            Node *const node,
            const int labelLen,
            const size_t inputLen )
```

Add the suffix indexes to the leaves of the suffix tree.

Depth-first search traversal of the suffix tree to set the suffix indexes of the leaves. The one of the internal nodes are set to -1. The function traverses the tree in a recursive way adding the suffix indixes to the leaves.

Definition at line 75 of file suffixTree.c.

4.22.2.3 applyExtensions()

```
void applyExtensions (
            unsigned *const input,
            int *const remainder,
            ActivePoint *const ap )
```

Apply the extensions to the tree.

This function is applied once for each input byte and it performs several operation in loop until remainder is 0:

- If there is not an edge from the current active node such that the label on the edge starts with the same character of the one in input, it creates a new leaf starting from the current node.
- Otherwise it walks down the tree until either:
  - The current char being processed is already on the edge, in that case the function returns. The remainder is not zero so the algorithm can keep track of the suffix the are missing in the tree. They will be generated in the next phases.
  - Otherwise an internal node is created because the input character is found in the middle of an edge.

Definition at line 108 of file suffixTree.c.

4.22.2.4 buildSuffixTree()

```
Node* buildSuffixTree (
            unsigned *const input,
            const size_t inputLen )
```

Returns the root of the suffix tree of the input sequence of bytes.

The function firstly initializes the active point and the root of the tree.
After that it performs n extensions of the suffix tree, where n is the length of the input string. Finally it adds the suffix indexes to the leaves of the tree and returns the root.
Basically the main function implementing the Ukkonen's algorithm is applyExtensions and is called a number of times equal to the length of the input string. Each time a new byte is processed.

Definition at line 42 of file suffixTree.c.

4.22.2.5 createInternalNode()

```
Node* createInternalNode (
            ActivePoint *const ap,
            HashChildren **const child,
            unsigned *const input )
```

Splits an edge by creating a new internal and a new leaf.

The functions needs to update/insert various part of the tree around the edge that must be split and the. This is done in different steps:

- Delete the old child from the children of the current node.
- Create internal node
- Attach the old child as new child of the new internal node, then update the old child label, so that the new start of the node is the actual plus the active length.
- Create new leaf from the internal node.

Definition at line 182 of file suffixTree.c.

4.22.2.6 createLeaf()

```
void createLeaf (
            ActivePoint *const ap,
            Node *const parent,
            unsigned *const input )
```

Creates a leaf of the tree and attach it to the parent passed as arg.

Parameters

in	*ap*	The current acrive point.
in,out	*parent*	The parent of the new leaf.
in	*input*	The input sequence.

Definition at line 214 of file suffixTree.c.

4.22.2.7 createNode()

```
Node* createNode (
            const int start,
            int *const end,
            Node *const root )
```

Creates a node with default initialization.

The default initialization of the node is performed by setting:

- The end, start passed as arguments.
- The root as suffix link.
- The set of children is NULL.
- The suffix index is set to -1.

Definition at line 165 of file suffixTree.c.

4.22.2.8 deleteChildren()

```
void deleteChildren (
            HashChildren ** children )
```

Frees the hash table that contains the children of a node.

Parameters

in	*children*	The hash table to be freed.

Definition at line 297 of file suffixTree.c.

4.22.2.9 deleteNode()

```
void deleteNode (
            Node * node )
```

Delete a node by free the memory for it and all of its children.

Parameters

in	*node*	The node to be deleted.

Definition at line 288 of file suffixTree.c.

4.22.2.10 getEdgeLen()

```
int getEdgeLen (
            Node *const node )
```

Returns the length of the label on the edge terminating in the input node.

Parameters

in	*node*	Node where the edge terminates.

Returns

The length of the label.

Definition at line 282 of file suffixTree.c.

4.22.2.11 printTree()

```
void printTree (
            Node * node,
            unsigned * input )
```

Depth-first traversal of the tree to print it.

Parameters

in	*node*	The root of the current subtree to be printed out.
in	*input*	The sequence in put.

Definition at line 307 of file suffixTree.c.

4.22.2.12 setSuffixLink()

```
void setSuffixLink (
            Node *const node,
            Node ** newest )
```

Sets a suffix link to a node.

Update suffix link if newest is pointing to a node, then reset newest for next iterations.
If newest is not null means that in the last iteration an internal node was created and now we set its suffix link.
The only node with suffixLink null is the root and all the other nodes have the root as default suffixLink.

Definition at line 243 of file suffixTree.c.

4.22.2.13 updateAP()

```
void updateAP (
            ActivePoint *const ap,
            const int remainder )
```

Update the active point.

Update the active point depending on whether the active node is root or not.

Definition at line 252 of file suffixTree.c.

4.22.2.14 walkDown()

```
int walkDown (
            ActivePoint *const ap,
            Node *const currentNode )
```

Walk down the tree onw level following one edge.

This function allows to walk down the tree quickly. In particulare if the active lenght is greater then the label length of the selected node, it can directly skip to that node by updating the active point.

Definition at line 268 of file suffixTree.c.

4.23 src/sources/unarith.c File Reference

Source file implementing an arithmetic coder compressor.

```
#include "../headers/unarith.h"
```

Macros

- #define BUF_BITS (8 ∗ sizeof(unsigned char))

  Number of bits of the byte buffer.

Functions

- Text arithDecoding (const Text input)

  Decode a sequence of bytes compressed by an arithmetic coder.
- void initDecoder (Decoder ∗const de, ByteBuffer ∗const inBuf, IOBuffer ∗const in, size_t inputLen)

  Initializes the decoder.
- unsigned decodeSymbol (Decoder ∗const de, Model ∗const model, ByteBuffer ∗const inBuf, IOBuffer ∗const in, size_t inputLen)

  Decode a sequence of bits and retrieve the symbol associated with the current interval read from the input sequence, that represents the interval in the model of probabilities.
- unsigned findChar (Decoder ∗const de, Interval ∗const interval, Model ∗const model)

  Selects the corresponding encoded symbol given a sequence of bits read. Sets also the interval of the decoded symbol and updates the model.
- void updateInterval (Decoder ∗const de, ByteBuffer ∗const inBuf, IOBuffer ∗const in, size_t inputLen)

  Updates the probability model.
- unsigned inputBit (Decoder ∗const de, ByteBuffer ∗const inBuf, IOBuffer ∗const in, size_t inputLen)

  Reads compressed bit and returns it.

4.23.1 Detailed Description

Source file implementing an arithmetic coder compressor.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.23.2 Function Documentation

4.23.2.1 arithDecoding()

```
Text arithDecoding (
            const Text input )
```

Decode a sequence of bytes compressed by an arithmetic coder.

This functions decompresses an input sequence of bytes using an arithmetic decoder and an adaptive model of probabilities.
It reads the sequence byte by byte and for each byte:

- Dcode the symbol using the actual probabilities..
- Update probabilities.

The probability model must be the same used for compressing the sequence.

Definition at line 60 of file unarith.c.

4.23.2.2 decodeSymbol()

```
unsigned decodeSymbol (
            Decoder *const de,
            Model *const model,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Decode a sequence of bits and retrieve the symbol associated with the current interval read from the input sequence, that represents the interval in the model of probabilities.

The operation is done by following three steps:

- Find char associated with the current state of the decoder and set the current interval of the symbol decoded.
- Update the current interval of the decoder with the read one.
- Update the probability model and check for other bits to read.

Definition at line 137 of file unarith.c.

4.23.2.3 findChar()

```
unsigned findChar (
            Decoder *const de,
            Interval *const interval,
            Model *const model )
```

Selects the corresponding encoded symbol given a sequence of bits read. Sets also the interval of the decoded symbol and updates the model.

Parameters

in	*de*	Pointer to the decoder.
out	*interval*	Pointer to the interval to be set.
in,out	*model*	Pointer to the model of probabilities.

Definition at line 160 of file unarith.c.

4.23.2.4 initDecoder()

```
void initDecoder (
            Decoder *const de,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Initializes the decoder.

The decoder is initialize so that the intervals is the total interval [0,1] and the read values are zero. The first 17 bits are read from the input.

Definition at line 119 of file unarith.c.

4.23.2.5 inputBit()

```
unsigned inputBit (
            Decoder *const de,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Reads compressed bit and returns it.

This function reads one compressed bit from the input buffer and returns it. When the end of file is reached, byt the fin variable of the decoder is set to 1, this functions returns the continuation bit to let the decoder finish the decompression.

Definition at line 222 of file unarith.c.

4.23.2.6 updateInterval()

```
void updateInterval (
            Decoder *const de,
            ByteBuffer *const inBuf,
            IOBuffer *const in,
            size_t inputLen )
```

Updates the probability model.

Parameters

in	*de*	Pointer to the decoder.
in,out	*inBuf*	Pointer to the buffer that keeps the current sequence of bits to decode.
in	*in*	Pointer to the sequence of bytes to decompress.
in	*inputLen*	The length of the sequence to decompress.

Definition at line 183 of file unarith.c.

4.24 src/sources/unbwt.c File Reference

Source file implementing the reverse Burrows-Wheeler transformation.

```
#include "../headers/unbwt.h"
```

Functions

- Text unbwt (Text input)

  Returns the reverse BWT of the input sequence of bytes.

4.24.1 Detailed Description

Source file implementing the reverse Burrows-Wheeler transformation.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.24.2 Function Documentation

4.24.2.1 unbwt()

```
Text unbwt (
              Text input )
```

Returns the reverse BWT of the input sequence of bytes.

The implementation follows the one based on the LF mapping, a function from the last column of the BWT to the first column. It is based on the lemma that the i-th occurrence of char c in the last column of the BWT matrix is the same exact character as the i-th occurrence c in the first column.
The algorithm starts reading the index of the first and of the sentinel characters. Then it calculates the frequencies of each character in the input sequence. After that it computes two arrays:

- One array that contains for each possible value i the number of values that are alphabetically smaller than i (mappings).
- One array that contains the LF mapping (links) calculated from mappings and frequencies.

Finally it outputs the original string from the LF mapping.

Definition at line 50 of file unbwt.c.

4.25 src/sources/unmtf.c File Reference

Source file implementing the reverse of the move to front transformation.

```
#include "../headers/unmtf.h"
```

Functions

- Text unmtf (const Text input)

  Reverses the move to front transformation of the input sequence of bytes.
- unsigned char searchSymbol (SymbolsList *const symbols, const unsigned char pos, SymbolsList **aux)

  Search the symbol in the list at the position specified in input.

4.25.1 Detailed Description

Source file implementing the reverse of the move to front transformation.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.25.2 Function Documentation

4.25.2.1 searchSymbol()

```
unsigned char searchSymbol (
            SymbolsList *const symbols,
            const unsigned char pos,
            SymbolsList ** aux )
```

Search the symbol in the list at the position specified in input.

The function implements a simple search of an element in a list of bytes. It outputs the corresponding element at the specified position and, at the same time, it stores in the double pointer aux the element before the one searched in the list, so as to make the next operation of moving ahead the element quicker.

Definition at line 72 of file unmtf.c.

4.25.2.2 unmtf()

```
Text unmtf (
            const Text input )
```

Reverses the move to front transformation of the input sequence of bytes.

The function is the symmetrical of the MTF transformation. In fact, it searches in the list of symbols the symbol which position is the one in the input, then it outputs the symbol and updates the list, moving the element in front of that. At the beginning, the symbols list is initialized as for the forward transformation.

Definition at line 40 of file unmtf.c.

4.26 src/sources/unzle.c File Reference

Source file implementing the reverse zero-length encoding.

```
#include "../headers/unzle.h"
```

Functions

- Text zleDecoding (const Text input)

  Reverse the zero-length encoding routine.
- void outputZeroes (unsigned char *runLen, unsigned char *out, size_t *index)

  Writes in the output array the sequences of zeroes encoded by the value pointed by runLen.

4.26.1 Detailed Description

Source file implementing the reverse zero-length encoding.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.26.2 Function Documentation

4.26.2.1 outputZeroes()

```
void outputZeroes (
            unsigned char * runLen,
            unsigned char * out,
            size_t * index )
```

Writes in the output array the sequences of zeroes encoded by the value pointed by runLen.

Parameters

in	*runLen*	Pointer to the binary number representing the number of zeroes of the original input.
in,out	*out*	Pointer to the output array where writes the zeroes.
in,out	*index*	Pointer to the index of the next cell to be written in the output array.

Definition at line 92 of file unzle.c.

4.26.2.2 zleDecoding()

```
Text zleDecoding (
            const Text input )
```

Reverse the zero-length encoding routine.

Parameters

in	*input*	The input sequence of bytes to be rebersed.

Returns

The reverse ZLE of the input.

Definition at line 33 of file unzle.c.

4.27 src/sources/util.c File Reference

Source file implementing a util interface for encoding bytes and managing files.

```
#include "../headers/util.h"
```

Functions

- unsigned char decomposeUnsigned (unsigned u, int n)
  Utility function to encode an unsigned.
- void encodeUnsigned (const unsigned u, unsigned char ∗const output, int outIndex)
  Stores in an array of unsigned char that represents ul big-endian.
- unsigned readUnsigned (unsigned char ∗const input, size_t n)
  Returns the unsigned encoded big endian by the 4 bytes of the input from location n.
- FILE ∗ openFileRB (char ∗const filename)
  Open a file for reading it.
- Text readFile (FILE ∗const file, long size)
  Reads at most size bytes from the file in input.
- FILE ∗ openFileWB (char ∗const filename)
  Opens a file for wiriting into it.
- void writeFile (FILE ∗const file, unsigned char ∗const buffer, long len)
  Writes the buffer into the file.
- long fileSize (FILE ∗const file)
  gets the number of bytes of the file.
- int compareFiles (FILE ∗const file1, FILE ∗const file2, long size1, long size2)
  Compare two files byte by byte.

4.27.1 Detailed Description

Source file implementing a util interface for encoding bytes and managing files.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.27.2 Function Documentation

4.27.2.1 compareFiles()

```
int compareFiles (
            FILE *const file1,
            FILE *const file2,
            long size1,
            long size2 )
```

Compare two files byte by byte.

Parameters

in	*file1*	The first file to compare.
in	*file2*	The second file to compare.
in	*size1*	The number of bytes of the first file-
in	*size2*	The number of bytes of the second files.

Returns

- 2 if the two sizes are different.
- 1 if the two files are equal.
- 0 if the two files are different.

Definition at line 150 of file util.c.

4.27.2.2 decomposeUnsigned()

```
unsigned char decomposeUnsigned (
            unsigned u,
            int n )
```

Utility function to encode an unsigned.

Parameters

in	*u*	The unsigned to decompose.
in	*n*	The byte of the decomposed unsigned.

Returns

Returns the nth big-endian byte of an unsigned int. (0 <= n <= 3)

Definition at line 34 of file util.c.

4.27.2.3 encodeUnsigned()

```
void encodeUnsigned (
            const unsigned u,
            unsigned char *const output,
            int outIndex )
```

Stores in an array of unsigned char that represents ul big-endian.

Parameters

in	*u*	The unsigned to encode.
out	*output*	Pointer to the output unsigned char where the result is stored.
in,out	*outIndex*	Index of the first element of the output array to be write.

Definition at line 39 of file util.c.

4.27.2.4 fileSize()

```
long fileSize (
            FILE *const file )
```

gets the number of bytes of the file.

Parameters

in	*file*	Input file.

Returns

The number of bytes of the file.

Definition at line 139 of file util.c.

4.27.2.5 openFileRB()

```
FILE* openFileRB (
            char *const filename )
```

Open a file for reading it.

Parameters

in	*filename*	The filename of the file to be opened.

Returns

The file construct of the opened file.

Definition at line 60 of file util.c.

4.27.2.6 openFileWB()

```
FILE* openFileWB (
            char *const filename )
```

Opens a file for wiriting into it.

Parameters

in	*filename*	The filename of the file to be opened.

Returns

The file construct of the opened file.

Definition at line 95 of file util.c.

4.27.2.7 readFile()

```
Text readFile (
            FILE *const file,
            long size )
```

Reads at most size bytes from the file in input.

Parameters

in	*file*	The file to be read.
in	*size*	The number of bytes to read.

Returns

The sequence of bytes of the size specified read in the file in input.

Definition at line 75 of file util.c.

4.27.2.8 readUnsigned()

```
unsigned readUnsigned (
            unsigned char *const input,
            size_t n )
```

Returns the unsigned encoded big endian by the 4 bytes of the input from location n.

Parameters

in	*input*	Pointer to the encoded unsigned int.
in	*n*	The index of the first byte of the encoded unsigned.

Returns

Returns the unsigned int encoded big-endian by the 4 bytes starting at location n.

Definition at line 47 of file util.c.

4.27.2.9 writeFile()

```
void writeFile (
            FILE *const file,
            unsigned char *const buffer,
            long len )
```

Writes the buffer into the file.

Parameters

in	*file*	The file to be written.
in	*buffer*	The buffer to be written into the file.
in	*len*	The length of the sequence of bytes to be written.

Definition at line 126 of file util.c.

4.28 src/sources/zle.c File Reference

Source file implementing a zero length encoder.

```
#include "../headers/zle.h"
#include <math.h>
```

Macros

- #define LOG2E 1.44269504089
  Log base 2 of e.

Functions

- Text zleEncoding (const Text input)
  Performs the zero length encoding.
- void encodeZeroRun (size_t runLen, Text *res)
  Encodes and stores the zero-run in input.

4.28.1 Detailed Description

Source file implementing a zero length encoder.

Author

Stefano Valladares, ste.valladares@live.com

Date

20/12/2018

Version

1.1

4.28.2 Function Documentation

4.28.2.1 encodeZeroRun()

```
void encodeZeroRun (
            size_t runLen,
            Text * res )
```

Encodes and stores the zero-run in input.

This function encodes a sequence of zeroes of length zeroRun in the big-endian representation of (zeroRun + 1), eliding the MSB.

Definition at line 92 of file zle.c.

4.28.2.2 zleEncoding()

```
Text zleEncoding (
            const Text input )
```

Performs the zero length encoding.

The main method of the file encodes a sequence of length N by compressing sequences of consecutive zeroes. Each one of that will be encoded in a binary number that represents the length of the zero-run. Each byte in input is encoded in an online fashion, written it down as soon as it is read, while each sequence of zeroes is encoded as soon as it ends by calling the function encodeZeroRun.

Definition at line 44 of file zle.c.

Index