Adaptive Huffman Encoding

Generated by Doxygen 1.8.10

Sun Oct 9 2016 04:46:19

Contents

Chapter 1

Adaptive Huffman CS3302-DE

Author: 140013444

Marker: Tom Kelsey

Date: 7th October 2016

Quick Start

```
$ echo -n "hello world" | ./huff -e | ./huff -d
>> hello world
```

Specification

This application was written as a solution for Practical 1 of module CS3302-DE at the University of St Andrews. The stated purpose of the pratical was to gain experience with a widely used compression algorithm.

The specification was as follows:

- Implement an Adaptive Huffman encoder (persistant tree is optional)
- Implement an Adaptive Huffman decoder (decoder optional)

Choosing not to implement any of the optional parts of the specification would limit the maximum grade to a 16. Choosing not to implement both optional parts of the specification would limit the maximum grade to a 13.

It was specified that any reasonable adaptive algorithm could be implemented.

It was also stated that the preferred implementation language was Java and that C++, C and Python are also acceptable.

Architecture

Project Directory Structure

```
.
 CMakeLists.txt
 Doxyfile.in
 README.md
 ext
  gtest
 inc
  Flags.hh
  buffer.h
   node.h
   tree.h
  src
   lib
```

```
 main.cc
test
    buffer_test.cc
    tree_test.cc
```

Top level overview:

- `CMakeLists.txt` defines the build process for the project.
- `Doxyfile.in` contains the configuration for the automatic documentation generation.
- `ext/` contains external library files
- `inc/` contains project header files
- `src/` contains project src files (binaries and executables)
- `test/` contains test files

File naming conventions are as follows:

- Internal header files use '.h' extension
- Internal source files use '.cc' extension
- Data structure unit tests use structure header name with '_test' postfix

Tooling Overview

Although the preferred language was Java, C++ was chosen for several reasons. These are listed in approximately descending order of priority:

Personal Growth: JH Team Project was democratically decided to be written in C++. As I have never previously had the opportunity to learn the language, this project seemed ideal for gaining some foundational experience.

Simplicity: Reading and writing to binary files and streams is abstracted over in the Java standard library. While this is useful for most purposes, productive use of these abstractions requires a familiarity with the standard library that I do not currently possess. C++, however, uses C primitives with which I have had previous experience.

Performance: C++ is generally a more performant langauge and should therefore allow for faster compression speeds.

Tooling Thanks to JH project preparation, I have become familiar with the wide variety of high quality C++ tools available.

A brief summary of tools used follows:

- **Build Tool:** CMake
- **Testing:** GoogleTest
- **Continuous Integration:** Travis CI
- **Documentation:** Doxygen

Implementation Approach

The algorithm is specifically dynamic and should process each symbol only once. For this style of task, `iostream`s are well suited.

Additionally, the wide range of unix tools (e.g. cat, echo, head, tail, xxd) provide utilities for inspecting and managing these types of streams on the command line. With this in mind, and the focus of the practical being on the algorithm and not the UI, a simple CLI was chosen. Any data stream can be piped into the program via `stdin` and the output simply dumps to `stdout`. To allow for the possibility of more elegant interfaces (and in the spirit of modular software design) the data structure is implemented as a standalone library and builds as such.

See the Usage section for more information.

Algorithm

This Adaptive Huffman encoding is a variant of the FGK algorithm (Knuth 1985).

The key data structure is a dynamic representation of the code tree with `encode` and `decode` functions that are defined as follows:

```
fn encode_symbol():
    x := read_symbol()
    if tree_ontains(x):
        output(x)
        Tree.acknowledge(x)
    else:
        output(path_of(NYT))
        output(path_of(x))
```

```
fn decode_symbol():
    n := root
    while !leaf(n):
        n := step(receive_bit())

    if is_nyt(n):
        symbol := receive_byte()
    else:
        symbol:= symbol_of(n)

     output(symbol)
     Tree.acknowledge(symbol)
```

The Tree is a binary tree structure with the following invariant:

```
All nodes can be listed in order of non-increasing weight with each node adjacent to its sibling.
```

Before updating node weights, it is checked if any node exists with the same weight higher up in the tree. If such a node exists, the current node is swapped with the highest && right-most node with the same weight. This means that the invariant is enforced and results in the heavier weighted nodes appearing higher in the tree.

From the `encode` and `decode` functions, we can see the data structure be dynamically maintained through the use of the `Tree.acknowledge()` which is defined here:

```
fn Tree.acknowledge(symbol):
    if Tree.contains(symbol):
        // output is noop when decoding
        output_path_to(leaf_wtih(symbol))
    else:
        output_path_to(nyt)
        output(symbol)
        split_nyt(symbol)

     node := leaf_with(symbol)

     while node != root:
         perform_swap()
         increment weight of node
         node := parent(node)
```

The C++ implementation of this algorithm stays fairly true to the psuedocode displayed here, though there are a few other considerations.

- Leaf node lookup
- Highest node in weight group lookup
- Symbol length

The lookup operations can be (and have been in this implementation) implemented in amortized constant time.

The approach for each lookup is independent.

Leaf Lookup

As each leaf is uniquely identified by its symbol, a mapping of symbols to leaf node pointers is simply maintained. All this requires is an insertion into the map whenever a new symbol is seen and then leaf lookup and the contains operation are both constant time map indexing.

It is worth noting here that the standard `map` object in C++'s standard library does not offer the performance profile described here. Instead it is ordered and the `unordered_map` object is required. This was discovered while profiling performance post-development and was luckily a simple fix. The performance increase was noticable for larger files but as the speed of compression was secondary to the compression rate in this pratical further performance profiling was not done.

Highest Common-Weighted Node Lookup

Performing ths lookup in constant time was slightly more interesting than simply including a mapping.

A doubly-linked circular list of nodes is used to define a weight class - with the head of the list being the highest (and right-most) in the tree.

Whenever a new node is added to the group (via a weight increment, tree insertion, etc.) it is placed to the rear of the list (constant time operation) and then, provided tree swaps also swap in the list, the order of nodes within the group is maintained.

With a mapping of weights to the heads of the corresponding lists, we can achieve constant time lookup for this operation.

Symbol Length

The current implementation supports only symbols of one byte in size. A change in symbol length may provide several changes in performance:

For example, an increase in symbol length would:

- Increase the maximum tree size (capitalizing on the lookup optimizations)
- Possibly better capture the information in certain encodings (such as UTF-16)

Evaluation

While the FGK variant was implemented due to prevalance in literature and of well documented implementations, other options were explored.

The strongest alternative was was proposed by Vitter. The data structure is very similar to FGK with subtle changes to the invariant:

```
For each weight w, all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w.
```

The implicit numbering is similar to the concept of order in FGK: The bottom-left node has the lowest numbering and it increases as we move up the tree.

The algorithm has a similar compression rate in the average case but is resilient against FGK's pathological case.

Both of these algorithms are extremely sensitive to errors. If a single bit is corrupted within the stream, the encoding of all following bits will be shifted resultingly. Several approaches can be employed to combat this, including segmentation of the data stream. This does however present challenges, as the tree has to be rebuilt for every new segment of the stream.

If a sufficiently dynamic tree was designed, such that the segmentation approach was feasible, then the resulting coding would be far more adaptive and would profit from clustering of symbols and similar phenomena that occur in certain information streams.

With more time this area be highly interesting to investigate further.

Usage Instructions

Building

The project uses cmake to handle the build process. As is standard when using cmake, the following command will build the project:

```
$ mkdir build && cd build && cmake .. && make
```

To cleanly remove the newly built project, simply delete the `build/` directory.

```
$ rm -rf build/
```

The generated makefile will contain specific targets for building only the tests, the project library or the `huff` binary executable.

Tests

The unit tests will be built automatically as part of the build process. To run the tests, run the following command within the `build/` directory:

```
$ ctest -VV
```

Usage

The main executable can provide its own usage instructions:

```
$ huff --help
```

but examples of compression and decompression are shown below

Encoding

```
$ cat input.txt | huff -e > compressed.txt
```

Decoding

```
$ cat compressed.txt | huff -d > decompressed.txt
```

Testing

Asserts are used to check the data structure invariants are maintained across operations. Additionally, various buffer structures and some simple end-to-end example uses are unit tested.

References

FGK Description (and psuedocode) Vitter's paper on Dynamic Huffman Codes (Vitter 1989)

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

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

Chapter 5

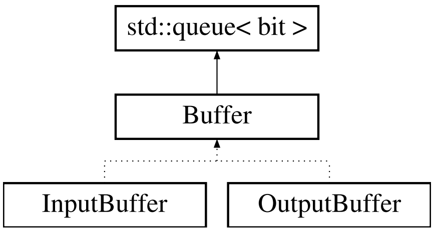
Class Documentation

5.1 Buffer Class Reference

Base class for bit/byte stream conversion via buffer.

```
#include <buffer.h>
```

Inheritance diagram for Buffer:



Public Member Functions

- int pop_byte ()

  Pops and returns a full byte from the buffer.
- void push_byte (byte input)

  Pushes a full byte to the buffer.

5.1.1 Detailed Description

Base class for bit/byte stream conversion via buffer.

Encapsulates the common behavior of the InputBuffer and OutputBuffer.

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

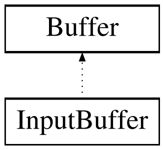
- /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/buffer.h

5.2 InputBuffer Class Reference

Buffers input bytestream and provides bitwise operations.

```
#include <buffer.h>
```

Inheritance diagram for InputBuffer:



Public Member Functions

- InputBuffer (std::istream &stream)

  Standard constructor.
- int receive_byte ()

  Obtain the next full byte from the buffer.
- int receive_bit ()

  Obtain the next bit from the buffer.

Private Attributes

- std::istream & stream

  The input bytestream.

Additional Inherited Members

5.2.1 Detailed Description

Buffers input bytestream and provides bitwise operations.

Data streams are typically bytestreams, but the compression algorithms require bitstreams. This class buffers up bytes from the stream and provides convenient bitwise accessors

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

- /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/buffer.h

5.3 Node Class Reference

Node class for Adaptive Huffman code tree.

`#include <node.h>`

Public Member Functions

- bool is_leaf ()

  Checks if the current node has any children.
- Node ∗ get_child (bool right)

  Gets the node in the given direction (left = 0, right = 1)
- int get_weight ()

  Get the Node's associated weight.
- char get_symbol ()

  Get the Node's associated symbol.

Private Member Functions

- Node ()

  Constructor for NYT Node.
- Node (char)

  Constructor for leaf Node.
- Node (Node ∗, Node ∗)

  Constructor for branch Node.
- void transmit_path (OutputBuffer &)

  Pushes the path to this node to the output buffer (left = 0, right = 1)
- void set_left (Node ∗)

  Releases the left node and takes ownership of the given node.
- void set_right (Node ∗)

  Releases the right node and takes ownership of the given node.

Private Attributes

- int weight

  Num occurrences for the symbol (or sum of children for internal)
- char symbol

  The symbol represented by this leaf (unused for internals)
- Node ∗ parent

  Pointer to the node's parent.
- Node ∗ group_next

  Pointer to the next node in weight group.
- Node ∗ group_prev

  Pointer to the prev node in weight group.
- std::unique_ptr< Node > **left**
- std::unique_ptr< Node > right

  Node owns its children.

Friends

- class Tree

  The tree structure manages and manipulates its nodes.

5.3.1 Detailed Description

Node class for Adaptive Huffman code tree.

The Tree performs adaptive Huffman encoding/decoding between the provided streams.Implementation prioritizes time over memory. Similarly, inexperience with C++ prevented splitting different node types into seperate classes. This means the tree likely has a substantially higher space complexity than could be accomplished with a nicer Node model which utilises inheritance to capture the behavior of Leaves, Branches and the NYT node.

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

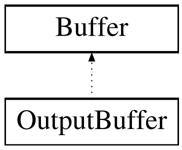
- /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/node.h

5.4 OutputBuffer Class Reference

Buffers output bytestream and provides bitwise operations.

`#include <buffer.h>`

Inheritance diagram for OutputBuffer:



Public Member Functions

- OutputBuffer (std::ostream &stream)

  Standard constructor.
- ∼OutputBuffer ()

  Pads to a full byte with 0 bits.
- void flush (bool=false)

  flush all full bytes from the buffer
- void send_byte (byte out_byte)

  Push a full byte into the buffer (may flush)
- void send_bit (bit bit)

  Push a single bit into the buffer (may flush)

Private Attributes

- std::ostream & stream

  The output bytestream.

Additional Inherited Members

5.4.1 Detailed Description

Buffers output bytestream and provides bitwise operations.

Data streams are typically bytestreams, but the compression algorithms require bitstreams. This class buffers up bits and provides convenient bitwise push functions

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

- /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/buffer.h

5.5 Tree Class Reference

Adaptive Huffman Code Tree.

`#include <tree.h>`

Public Member Functions

- Tree (std::istream &, std::ostream &)

  Standard constructor.
- bool contains (char)

  Checks the tree for a given symbol.
- void process_symbol (char, OutputBuffer &)

  Encodes a single symbol and update structure.
- void encode ()

  Begins encoding of the input stream.
- void decode ()

  Begins decoding of the input stream.

Private Member Functions

- void update_weight (Node ∗)

  Increments leaf weight or recalculates branch weight.
- void change_weight (Node ∗, int)

  Changes node weight, while maintaining groups.
- void perform_swap (Node ∗)

  Swaps the node with the heighest weighted in group.
- Node ∗ get_root ()

  Provides a ptr to the root node.
- Node ∗ get_weight_group (int weight)

  Provides a ptr to the head of the weight group.
- void set_root (Node ∗root)

  Takes ownership of the new node (releases old)

Private Attributes

- std::istream & input

  Buffered input stream.
- std::ostream & output

  Buffered output stream.
- Node ∗ nyt

  Maintained pointer to the NYT node.
- std::unique_ptr< Node > root

  Tree owns root to provide auto cleanup.
- std::unordered_map< int, Node ∗ > groups

  Mapping from weight to heighest node of weight.
- std::unordered_map< char, Node ∗ > leaves

  Mapping from symbol to representing leaf.

5.5.1 Detailed Description

Adaptive Huffman Code Tree.

The Tree performs adaptive Huffman encoding/decoding between the provided streams.

Overview

As the focus of the practical was to implement an Adaptive Huffman algorithm, the Tree had to be able to effectively compress a stream of data while making as few assumptions about its characteristics as possible.

Exisiting Works

An examination of the existing works on Adaptive Huffman algorithms reveal two well known algorithms:

- `Vitter's`
- [FGK](TODO: add FGK Source)

Chosen Approach

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

- /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/tree.h

Chapter 6

File Documentation

6.1 /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/buffer.h File Reference

Buffer structures for converting between bit/byte streams.

```
#include <iostream>
#include <queue>
```

Classes

- class Buffer
  Base class for bit/byte stream conversion via buffer.
- class InputBuffer
  Buffers input bytestream and provides bitwise operations.
- class OutputBuffer
  Buffers output bytestream and provides bitwise operations.

Typedefs

- typedef unsigned char **byte**
- typedef bool **bit**

6.1.1 Detailed Description

Buffer structures for converting between bit/byte streams.

Author

140013444

Date

7 Oct 2016 Three classes are provided.

- A Buffer base class, providing the common functionality of a simple bit/byte queue
- A specialized InputBuffer which wraps an istream of bytes and ptrovides a queue of bits/bytes.
- A specialized OutputBuffer which wraps an ostream of bytes and ptrovides a queue of bits/bytes.

There's plenty of room for optimization and improvement here but, as the focus of the pratical should be on the compression algorithm, the implementation is pretty barebones.

6.2 /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/node.h File Reference

Node class for CS3302-DE Adaptive Huffman coding.

```
#include <memory>
#include <iostream>
#include "buffer.h"
```

Classes

- class Node

  Node class for Adaptive Huffman code tree.

6.2.1 Detailed Description

Node class for CS3302-DE Adaptive Huffman coding.

Author

140013444

Date

7 Oct 2016

6.3 /Users/jac32/Documents/Assignments/CS3302-DE/Adaptive-Huffman/inc/tree.h File Reference

Main Tree structure for CS3302-DE Adaptive Huffman coding.

```
#include <iostream>
#include <unordered_map>
#include <memory>
#include "buffer.h"
#include "node.h"
```

Classes

- class Tree

  Adaptive Huffman Code Tree.

6.3.1 Detailed Description

Main Tree structure for CS3302-DE Adaptive Huffman coding.

Author

140013444

Date

7 Oct 2016