Program Specification for *Colden*

By Holden Lee and Cole Graber-Mitchell

High Level

Colden is a GUI-based file compressor and decompressor that allows single files to be, unsurprisingly, compressed and decompressed. When run, a user interface opens that has two top-level tabs, "Compress" and "Expand." Each allows the user to specify an input file and an output location. Functionally, the program only works with its own proprietary file format, denoted by the suffix `.112`. A file compressed with the program, powered by a Huffman tree encoding, is losslessly made smaller, and is able to be expanded into an absolutely bit-perfect copy of the original. The algorithm we use adds a small table to the beginning of the file that, while adding some fixed size to each compressed file, should allow the actual body of the file to be compressed to a fraction of its original size. In files with a lot of repetition, compression will be better.

File compression is an important part of the modern computing world. Whenever servers send data to clients or files are archived into long-term storage, compression is used to reduce the size of data while still allowing the original data to be recovered. There exist a wide variety of compression algorithms and specifications that more or less fill the field, and this project doesn't attempt to replace any or even work within existing standards. Instead, our goal is to experiment with the Huffman algorithm to allow the compression of single files losslessly.

The user will be able to select files from the system. They will also be able to click on the tabs to alternate between compressing and expanding files. There will be buttons to prompt the user to select a file from the system, to set an output location and file, and to execute the compression/decompression. There is also a checkbox to allow the user to create a visual representation of the Huffman tree for the compression, as well as a button to choose a location for the file containing the tree. The bottom of the GUI contains a log, which updates the user on progress and whether any errors have occurred.

The program will respond by either executing a compression/expansion and updating the log accordingly, or updating the log with the appropriate error messages if the specified action cannot be performed.

Low Level

`public class Main extends Application { ... }`

This is the entry point of the program, as required by the specification. It uses the JavaFX framework to drive the GUI, and so this class overrides the `start(Stage)` method to construct the GUI as well as delegate to the other classes.

During the lifetime of the program, only one instance will exist, although we don't see that instance. Its creation is handled entirely by the JavaFX framework. For this reason, we do not write a constructor method. Instead, we use the `start` method for any initialization we need.

There are two member variables of the class: `Compress compress` and `Expand expand`. These variables hold instances of their respective classes and are used by the objects `Tab compressTab` and `Tab expandTab` to draw those pages of the application and they are initialized in the `start` method.

The `Main` class has two functions: `public void start(Stage)` and `public static void main(String[])`. The former contains the initialization and drawing code. The `Stage` argument is JavaFX's class that controls what is shown in the window. The latter, `main`, is the traditional entry point of Java programs. In our case, because our application uses JavaFX, this method is irrelevant in most cases. In case it ever is called, it merely delegates to the JavaFX framework immediately.

`public class ColdenTab { ... }`

This class defines the layout of the GUI, and provides a framework for the implementation of the Huffman compression algorithm. `ColdenTab` is never actually used, except as the parent for `Compress` and `Expand`.

`ColdenTab` has a number of member variables, which can be broken down into 5 groups. `Stage stage` and `GridPane grid` are solely for the GUI, and allow the other elements to be organized neatly. The input fields consist of a `File input`, 2 `Button` objects to select and deselect a file location for the input file, and a `TextField` to display the file location. An identical group of fields exist for the purpose of determining the output file location. There are exists an identical group of fields to set a location for a .pdf of the Huffman tree generated during compression. Additionally, there is also a CheckBox field which allows the user to dictate whether a separate tree file will be generated at all.

The constructor for `ColdenTab` takes in only a `Stage stage`, which forms a background for the GUI. The constructor calls a method, `makeGUI()`, which defines all the fields, and adds them to `GridPane grid`, which allows them to be ordered cleanly within the window. `makeGUI()` also tells the program how to respond when the buttons are clicked, and defines the procedure for prompting the user to select file locations. It takes in a `Stage stage`, and has no return type.

`ColdenTab` has a few methods. `doOperation()` is just a method stub (with no parameters or return), to be overridden in subclasses `Compress` and `Expand`. The method `render()` takes no input and simply returns the `grid`, so that `Main` can add that, and the objects contained within it, to the window. `log(String message, Level prio)` is used to print out any errors or warning that are generated while executing a compression/expansion. It takes in the error message, and a `Level prio`, which indicates the type of message to display (informational, warning, error, etc.). It returns nothing.

`public class Compress extends ColdenTab { ... }`

This class, as a subclass of `ColdenTab`, has all the properties and functions of its parent class, specified above. The only difference between `Compress` and `ColdenTab` is that `Compress` overrides the method `doOperation()`, which now executes the Huffman compression. It will

create an instance of `Huffman`, which it will use to generate character frequency tree and create a new, compressed file at the specified output location.

`public class Expand extends ColdenTab { ... }`

This class is also a subclass of `ColdenTab`. It behaves nearly identically to its parent, but like `Compress`, `Expand` overrides `doOperation()`. However, instead of using an instance of `Huffman` for compressing a file, it uses the class to expand a file already compressed by the Huffman algorithm.

`public class Huffman { ... }`

This is a data structure that represents a built Huffman tree. It can be built algorithmically from input data during compression or deserialized directly from the header of a file during expansion. During the course of the program's lifetime, infinitely many instances can be used: one corresponding for each time a file is compressed or expanded.

As a representation of a Huffman tree, `Huffman` contains one private member variable: a reference to the top level node of the tree. This variable is an instance of an internal, private class `Node` that has references and its left and right children, as well as a weight and list of node values. The values are all of the bytes that the children of this node refer to. If the list has a length of 1, then this node is a leaf node and refers to a byte itself. The weight is a representation of how frequent that byte appears in the input data. The top level node reference variable on the `Huffman` class is private to restrict information about the internals of the class and to prevent other classes from invalidating invariants necessary to the success of the compression and expansion algorithms.

Since the only data of the Huffman tree is private, a few public methods are exposed to facilitate its usage. The first is the constructor, which takes a byte array and uses it to build the Huffman tree. The second is a quasi-constructor: a static method on the `Huffman` class that returns an instance of itself deserialized from the header of a compressed file. This function, `public static Huffman deserialize(byte[])`, is used during expansion and takes a byte array that holds the serialized Huffman tree. The next function, `public byte[] serialize()`, serializes `this` into bytes, allowing it to be included in the header of the compressed output file. The remaining two functions, `public byte[]`

`compress(byte[])` and `public byte[] expand(byte[])`, use the built Huffman tree to compress and expand the input byte streams and return the results.

`public class Artifact { ... }`

This class represents a file that can be written and read by the program. It can be constructed from bytes (during expansion) or from a tree and a compressed body of bytes. It has getter methods for different parts of the file, `getSerializedTree()` and `getBody()`. Both of these methods return byte arrays of their respective parts. It also has a method `writeToPath(path)` that writes out the file. It has another method, `writeBytes(path, bytes)` that simply is a utility method to write the given byte array to the given path. Internally, when building a new file, it first serializes the tree, then constructs the header around it: the magic number in the beginning the the end-of-header marker at the end. Then, it appends the compressed body to that array and stores it. When parsing from a byte array, the class checks that the magic number is correct and then searches through the header to find the end-of-header marker, recording its position. These two fields, `bytes` and `markerIndex`, hold all of the required information for parsing: the raw data of the file and the index to the first byte of the marker sequence.

`public class BitArray { ... }`

The BitArray class represents an array of individually addressable bits. It is backed by an `ArrayList<Byte>` behind the scenes, allowing it to store a group of bits in the most easily addressable and smallest memory footprint possible. This class is necessary for reducing program complexity and moving all of the actual bitfiddling into a designated place. The class exposes three types of methods: getters and setters of indexed bytes, push and pop operations on various sizes, and reading multiple primitives from the Array. In addition to that, it has three constructors: one default constructor, one that can take an initial bit capacity, and one that copies another BitArray. Lastly, the class has a method to append a whole other BitArray to the current one.

The `get` and `set` methods are simple: they use a bitIndex to either get or set the specified byte. When setting a bit, an exception is thrown if its past the end of the array.

The push methods, made in three variants (bit, byte, int), allow those number of bits to be added to the end of the Array. These grow the internal backing if necessary and add the bits.

The read methods (single bit (get), byte, and int) read that number of bits from the Array. One of them, `readInt`, needs to be aligned at a byte boundary because that is the only time it is used in the program.

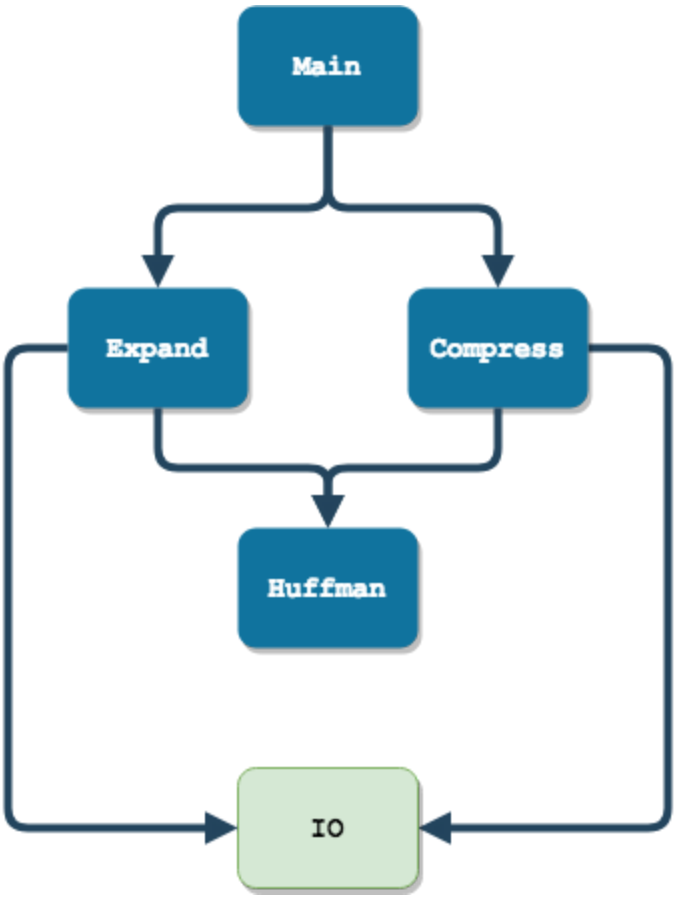
Other methods, like `fromBytes` and `toArray`, allow the BitArray to be converted to a more interoperable type for writing to or reading from disk.

`public class Graph { ... }`

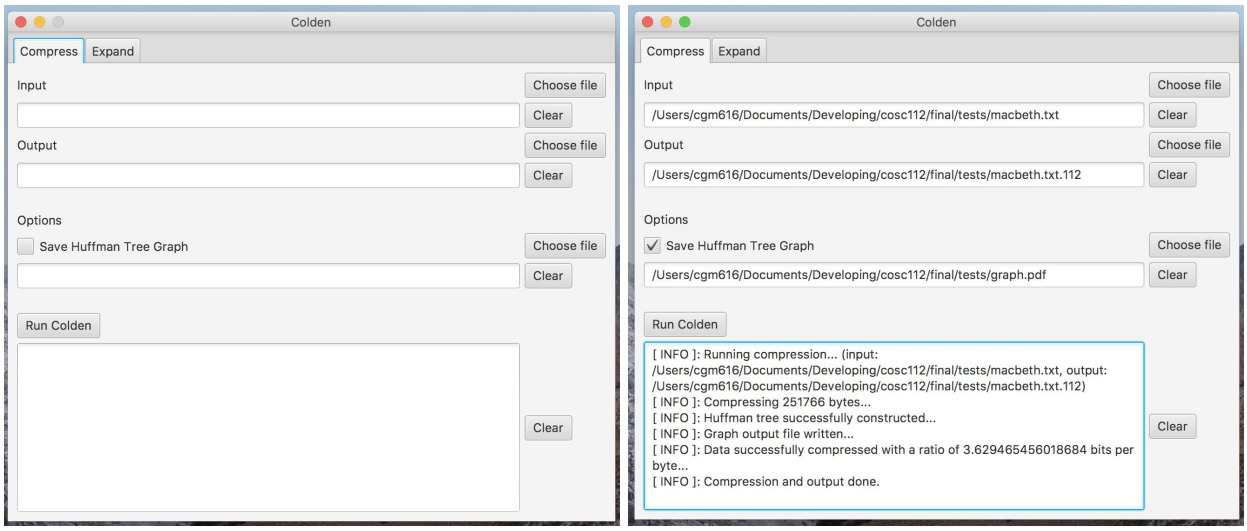
This class is charged with generating visual representations of the Huffman tree created during compression. It has only one field, a `Huffman tree`, and its constructor takes in a `Huffman`, which is assigned to `tree`. `Graph` has only one method, `write(File file)`, which takes in a `File` object and uses an external program to draw the graph in a .pdf file. It has no return type.

Class diagram

See class descriptions for a more detailed explanation of how the classes work together. This is simply a high level view, and leaves out many classes.



Screenshots



Left: program upon opening

Right: program after compressing a document