DEWK Compression

Willy Xiao, Kevin Eskici, David Herman, Eamon O'Brien

Demo video: http://youtu.be/tjv-niA6NAk

Overview:

We implemented a super-compressor that takes in a file, assesses the performance of several compression algorithms, and then compresses the file using the optimal algorithm. The four compression algorithms that we used were Fibonacci Encoding (Fib), Huffman Coding, Sequitur (Nevill-Manning algorithm), and Lempel-Ziv-Welch Encoding (LZW).

Because our primary goal was to test different algorithms and compare them, we did not have specific benchmark goals in advance for each algorithm. We were pleasantly surprised by the performance of some algorithms and disappointed with others. The most common algorithms chosen by our super-compressor are Huffman and LZW, with Fib working best for files with small integers. Sequitur, while functioning, doesn't output meaningfully compressed files (and for small to medium files, the compressed file is larger than the original) and only works for text files.

Initially we planned to conduct a study to compare the efficiency of each algorithm for various types of files. We quickly realized that, in addition to file type, the internal composition of the file has a substantial impact on which algorithm is appropriate. Instead of conducting the study, we decided to implement an extension that created an estimate function for each algorithm. When using our super-compressor, we then use each algorithm's estimate function to compare the algorithms and determine which one is most appropriate to the specific file being compressed. This extension makes our project substantially more adaptable in the future. First, it enables the easy addition of new algorithms. Had we based our super-compressor on a comparative study of file types, large portions of code would need to be altered to add a new algorithm. With estimate, the comparison method would be included within the new algorithm and is abstracted out of our core super-compressor program. Second, abstracting estimate allows us to include algorithms in our super-compressor that only work for some file types. If we want to implement an algorithm in the future that works exclusively for text files (some variants of sequitur for example), we can simply add a condition in that algorithm's estimate function that checks whether the file in question is of the appropriate type.

Once we finished our initial goals, we implemented an extension that allows our super-compressor to take in directories and determine the proper form of compression for the files contained within. The directory compressor then calls the appropriate algorithm on each file in the directory and concatenates the results into a single compressed file.

Planning:

We originally planned to have all of the algorithms done by Checkpoint 1 and proceed to work on the super compressor and directory compression plus implement a couple more algorithms after. Unfortunately, we underestimated how difficult compression was and various problems caused LZW and Sequitur to take an extra week and a half to finish as well as an extra few days for Huffman. We anticipated that Willy would finish Fibonacci coding quickly, so he was able to get a head start in setting up the framework for our super compressor and directory compression. Overall we were able to get all the functionality we intended (all algorithms done*(see reflection for sequitur) with the super compressor working), as well as implement directory compression as an extension. Due to time constraints we did not hit our additional extension goal of adding more algorithms, though we are thrilled that we got just about everything plus directory compression working!

In the report folder you can find our Final (Tech) and Draft Specs- surprisingly we actually were able to stick to our spec for the majority of the project and implement the described functionality. We annotated the final spec to show things we took out (in red) and new things we ended up with (in blue).

Design and Implementation:

Helpers --

Helpers abstracts common functions used in all of the different compression algorithms and for super-compressor. This was not in the original draft spec, but was necessary from the beginning. Any miscellaneous function that was needed more than once was included in here.

Some common ones are : start_compress and start_decompress which returns a file_in and a file_out to read and write from or helpers.freq_list which returns a frequency list of sample_size from the file.

Fib --

Fibonacci encoding is probably the most straight-forward compressor and is easily implemented.

It encodes any integer between 0 - 255 as the bits where each digit of the bit represents that index in the fibonacci sequence. Then an extra 1 is appended at the end which means any time two 1's appear in a row, the end of a character is represented. e..g "11011" means 5 * 1 + 3 * 1 + 2 * 0 + 1 * 1 = 9.

Fibonacci is the fastest compression mechanism because the codes aren't determined by the size or entropy of the file. In fact, it's just a variable-length way of representing the characters 0 - 255.

The implementation went smoothly and allowed Willy to work on other parts of the

project.

LZW --

Lempel-Ziv-Welch is also probably one of the most common compression algorithms which is easily implemented. The pseudo-code is a little bit complicated and it's explained at the top of code/algs/lzw.py. Essentially as files get larger, compression gets better because codes are generated as new sequences are discovered.

One downside to LZW is that it effectively has to run through each file twice (just like sequitur and huffman, substantially slowing down compression). This occurs because it must first generate the codes, determine the length of each integer that will be written and then run through the file again writing the integers.

Sequitur --

Sequitur iterates through a file and creates a dictionary of all diagrams (adjacent character/rule pairs). It then continually loops through the original file and replaces all repeated diagrams with a rule (an integer), until there are no repetitions of character pairs, rule pairs, or character-rule pairs. In addition to compressing files, Sequitur has potential applications for the statistical analysis of large texts, books, and more. Because it creates a hierarchical representation of repetitions, it allows the user to observe and analyse patterns in a way that many other compression algorithms do not. Potential future implementations could parallelize the algorithm to improve runtime.

Huffman Coding-- Implementing Huffman went a lot better than expected, largely due to the amount of abstraction involved. We wrote compress and decompress assuming we already had a dictionary of bytes and their respective "codes", and later went on to implement the functions necessary for building that dictionary such as frequency list, building a huffman tree from that frequency list, and then adding the codes to the tree. While all this went well there were some issues with our original attempt at implementing huffman which are described below in the reflection section.

Testing was done through test.py found in the *super* folder- by giving it a file name and directory the function will perform various tests on each part of our project.

Reflection:

Fib -- In retrospect, fib is not really much of a "compression" algorithm, but just another way of representing character as variable-length rather than constant-length. This was also a great decision to have somebody implement an easy algorithm, because it means that person can deal with the file/io, helpers. study, super etc.,

If there was someone dedicated to those other modules but didn't have to code any algorithms, it would also be very difficult to decide what abstractions are needed.

The algorithm itself could be improved to be more similar to huffman in whcih instead of encoding integers, there's a dictionary at the top of the file. Then the most commonly represented characters are given the shortest codes. This would be almost exactly like huffman but less effective. Essentially, fib should not be used for compression...And it shows because super-compressor almost never chooses it.

LZW --
LZW was a little bit tricky to implement at first, that's why we had to get some other group members to help out.

In the end, it's a beautiful algorithm and the final implementation of it went smoothly. The one problem that this runs into is that during the implementation of LZW, usually a maximum bit length is provided, which limits the number of potential codes. We, however, did not implement this maximum bit - length which means on a big file the codes could get arbitrarily large in which case it would not compress very well. This is easily done by providing a limit and would be one of the first improvements to our project.

Huffman Coding -- While the bulk of huffman was implemented smoothly, there were a few small but extremely significant issues that caused some trouble. The first was forgetting that the call to file_out.write added padding of zeros to make sure that the size of the output file was a multiple of 8 bytes (since each bit is represented as a 0 or 1 string in the intermediate step). Once this issue was discovered, it was resolved by adding a byte to the end out the output file that represented the number of zeros added for padding in the second to last bit, and using seek/set to get the necessary information at the start of decompression and get the position back to a point after the header. Another problem was that the implementation of huffman was not working well on large files, which was found to be because we were using a byte to represent the frequency of each byte in the header, but forgot that the frequency could exceed 255. This issue was solved by using 4 bytes to represent the frequency, though this comes at the cost of a much bigger header which means small enough files are more likely to get bigger.

Sequitur -- Implementing this was a lot more difficult than expected. Sequitur works for *most* files but does not work for non-text files and doesn't work for some text files that contain certain characters. There is also a bug in the file reading implementation (not the algorithm itself) that is triggered when sequitur is run twice on the same file without deleting the previously compressed files. We believe it has something to do with the files not being closed properly but didn't have time to find it and fix it. We found some of these bugs after testing on a variety of files late in the process. As a result, we excluded sequitur from the super-compressor, although it contains all the code necessary to be incorporated when this one bug is fixed. There are also several other barriers to full functionality. First, the algorithm requires relatively large files to exhibit any compression at all (due to the size of the dictionary being written into the compressed file). We

found that for files around 3 MB in size, sequitur compressed files begin to be smaller than the original files. To improve this in the future, we could experiment with different ways of representing the dictionary in the compressed file, or even finding a way to compress the dictionary itself further. The second issue relates to the runtime of the decompress function. We had two iterations of the Sequitur algorithm. Our first implementation had severe efficiency problems, as it was reading entire files into RAM for the purpose of traversing and removing all repetitions. In the second implementation, we substantially reduced the runtime for compressing large files by writing the compressed file byte by byte. Unfortunately, because sequitur must loop through the entire file to replace rules with the digrams they represent, decompress needs to load the entire file into RAM, substantially reducing running speed. Thus, we have hard coded a maximum file size into the estimate function for sequitur. If you want to test sequitur on a large file, be prepared for very long wait times. The compress function is much faster, meaning that you can observe the smaller size of the compressed files, you just cannot decompress them at this time. In the future, one way to fix this problem would be to create intermediate files in which the replacement of rules would occur, obviating the need to store the whole file in memory. We were unable to do this because of time, but it offers potential for future optimization.

Language Choice -- If we were to go back and do the project again, we probably would choose to do implement everything in C rather than in Python. This largely for the obvious reason that everything would run much faster in C, as well as the fact that we would not need to have an intermediary step while compressing and decompressing where we called the C programs reader and writer. Nonetheless we think that working in Python was a good experience as none of us had used it before and it forced us to learn more CS :)

Individual Contributions:

As a result of our modularization for the different compression algorithms, at the start of the project we were all able to start working on one individually (WIlly - Fibonacci, Kevin - Huffman, David - Sequitur, Eamon - LZW). Since Fibonacci was relatively easy to implement, Willy finished quickly and was able to spend much time developing our interface by contributing to helpers.py (with Kevin), as well as doing most of the super compressor and directory compression (this includes the study.py and estimation functions for each algorithm). Sequitur proved to be much harder to implement than anticipated, so that took up most of David's time (Kevin also helped debug) and he spent up until now trying to get it to work on non-text files but to no avail. There was some trouble with the original implementation of LZW, but with help from Willy and Kevin, Eamon was able to get it done. Kevin and Willy handled the earlier writeups, and completed this one with David while Eamon edited the video.

Lessons Learned:

Abstraction is your friend- Throughout the project we were incredibly thankful that we decided to abstract many details away with reader, writer, our modularization of the different algorithms, and especially helpers. Specifically, we learned not to wait until we find the need to copy/paste code to abstract it, but instead to plan ahead and know what interfaces we will be working with from the beginning.

Bit manipulation is hard- Especially when you have to deal with padding and reading, writing, etc. There was a point when our decompressed file was off by one or two bits and it highlighted the importance of slowing down and reasoning about each step of your code.

Type control - Unlike OCaml, Python is dynamically typed, which meant that we were not able to count on the compiler to catch errors where we are returning something of the wrong type. This meant that we would have to assert that certain things were of the type that we expected instead of being able to rely on the compiler.

Next Steps:

There are a few things we would work on if we had more time, the first of which is improving the estimate function for our algorithms since the super compressor relies on that for choosing which is used. Additionally we would try to implement a couple more algorithms as options for super to choose from, as well as getting sequitur to work on non-text files if possible. Next we would work on a better way of handling our “intermediary step” that results from using reader & writer and potentially find a way to get rid of it completely. Finally we could try to optimize for speed instead of just the size of the compressed file, though this would require rewriting our whole program in C (Note: this would make the intermediate step unnecessary too so it would kill two birds with one stone)

Advice for future students:

THINK before you write code! If there is functionality that many files and group members are going to need throughout the project, abstract it away from the start rather than waiting until you need to copy/paste it or have different people spend time writing the same thing. Having a well defined interface from the start will also help ensure that everyone is respecting the same invariants, which is critical in things like our super-compressor which calls functions from many modules written by different people.