38.7 Summary

Compression Model #1: Algorithms Operating on Bits. Given a sequence of bits B, we put them through a compression algorithm C to form a new bitstream C(B). We can run C(B) through a corresponding decompression algorithm to recover B. Ideally, C(B) is less than B.

Variable Length Codewords. Basic idea: Use variable length codewords to represent symbols, with shorter keywords going with more common symbols. For example, instead of representing every English character by a 8 bit ASCII value, we can represent more common values with shorter sequences. Morse code is an example of a system of variable length codewords.

Prefix Free Codes. If some codewords are prefixes of others, then we have ambiguity, as seen in Morse Code. A prefix free code is a code where no codeword is a prefix of any other. Prefix free codes can be uniquely decoded.

Shannon-Fano Coding. Shannon-Fano coding is an intuitive procedure for generating a prefix free code. First, one counts the occurrence of all symbols. Then you recursively split characters into halves over and over based on frequencies, with each half either having a 1 or a 0 appended to the end of the codeword.

Huffman Coding. Huffman coding generates a provably optimal prefix free code, unlike Shannon-Fano, which can be suboptimal. First, one counts the occurrence of all symbols, and create a “node” for each symbol. We then merge the two lowest occurrence nodes into a tree with a new supernode as root, with each half either having a 1 or a 0 appended to the beginning of the codeword. We repeat this until all symbols are part of the tree. Resulting code is optimal.

Huffman Implementation. To compress a sequence of symbols, we count frequencies, build an encoding array and a decoding trie, write the trie to the output, and then look up each symbol in the encoding array and write out the appropriate bit sequence to the output. To decompress, we read in the trie, then repeatedly use longest prefix matching to recover the original symbol.

General Principles Behind Compression. Huffman coding is all about representing common symbols with a small number of bits. There are other ideas, like run length encoding where you replace every character by itself followed by its number of occurrences, and LZW which searches for common repeated patterns in the input. More generally, the goal is to exploit redundancy and existing order in the input.

Universal Compression is Impossible. It is impossible to create an algorithm that can compress any bitstream by 50%. Otherwise, you could just compress repeatedly until you ended up with just 1 bit, which is clearly absurd. A second argument is that for an input bitstream of say, size 1000, only 1 in 2^499 is capable of being compressed by 50%, due to the pigeonhole principle.

Compression Model #2: Self Extracting Bits. Treating the algorithm and the input bitstream separately (like we did in model #1) is a more accurate model, but it seems to leave open strange algorithms like one in which we simply hardcode our desired output into the algorithm itself. For example, we might have a .java decompression algorithm that has a giant `byte[]` array of your favorite TV show, and if the algorithm gets the input `010`, it outputs this `byte[]` array.

In other words, it seems to make more sense to include not just the compressed bits when considering the size of our output, but also the algorithm used to do the decompression.

One conceptual trick to make this more concrete is to imagine that our algorithm and the bits themselves are a single entity, which we can think of a self-extracting bit sequence. When fed to an interpreter, this self-extracting bit sequence generates a particular output sequence.

Hugplant Example. If we have an image file of something like the hugplant.bmp from lecture, we can break it into 8 bit chunks and then Huffman encode it. If we give this file to someone else, they probably won't know how to decompress it, since Huffman coding is not a standard compression algorithm supported by major operating systems. Thus, we also need to provide the Huffman decoding algorithm. We could send this as a separate .java file, but for conceptual convenience and in line with compression model #2, we'll imagine that we have packaged our compressed bit stream into a `byte[]` array in a .java file. When passed to an interpreter, this bitstream yields the original hugplant.bmp, which is 4 times larger than the compressed bitstream + huffman interpreter.

38.6 LZW Compression

Next
38.8 Exercises

Last updated 1 year ago