DD2440 Advanced Algorithms
Lecture 14: Data Compression

Lecturer: Johan Håstad
Scribe: Björn Dagerman

8 December 2014

1 Introduction

The basic data compression problem is compressing long strings into shorter strings. It's important to consider that no compression algorithm can compress all strings. This can proved by a counting argument. Some strings get shorter while some get longer.

2 Kolmogorov complexity

Definition 1. *K(x) = The length (in bits) of the shortest program outputting x.*

Does it matter what language the program is written in? For instance, lets consider the languages C and Java. The relationship between them can be expressed as:

$$K_C(x) \leq K_{Java}(x) + B'$$
$$K_{Java}(x) \leq K_C(x) + B$$

where B, B' are two constants. Let us motivate the second inequality, the first being similar. If given a very efficient C-program that outputs x, one could write a C-compiler and simulator in Java that:

1. Takes a C-program as input.
2. Checks that it's correct.
3. Runs it on an empty input.

Assuming that there is a fixed size Java-program that checks correctness of and simulates C-programs, the size of this resulting program is the size of the C-program added with an absolute constant B.

We conclude that the only difference in Kolmogorov complexity defined by different programming languages is an additive constant. The same argument can be generalized for many different notions of programs.

It's sometimes more, or less convenient, to consider programs with or without input. However, running a program on a particular input is essentially the same as running it using a program without inputs. In such cases, the input can be hard coded and thus the only difference in length is n, the size of the input. Therefore, for programs without inputs (using a hard coded input) the total size of the program is the size of the old program plus n.

Theorem 1. *For any compression algorithm A:*

$$\mathit{Length}(A(x)) \geq K(x) - C_A$$

for some constant C_A independent of x.

Proof. A program that outputs x is given by the decompression algorithm D_A together with $A(x)$. The size of the program is essentially the size of the code for D_A and the code for $A(x)$. Therefore, the total size is $\mathit{Length}(A(x)) + C_A$ for some constant C_A, which is independent of x. Thus, we have a program of size $\mathit{Length}(A(x)) + C_A$ that outputs x and by the Kolmogorov definition we have $K(x) \leq \mathit{Length}(A(x)) + C_A$ and the theorem follows. □

Theorem 2. *For a random x:*

$$K(x) \geq |x| - 2 \text{ with probability } 3/4$$

where $|x|$ is the length of x.

Proof. Suppose $|x| = n$. There are at most $2^{(n-2)}$ strings of length $n-2$, and at most this many programs. Each program produces one output. This yields at most $2^{(n-2)}$ strings of length n with $K(x) \leq 2^{(n-2)}$. There are 2^n strings of length n. Thus there are at least $2^n - 2^{(n-2)} = 3/4 \cdot 2^n$ strings that requires length $\geq n-2$ programs to be generated. □

Let us consider the following classical paradox: *"The smallest integer that cannot be described in the English language with less than a thousand words"*. How can we define "describe"? A reasonable definition could be: *"can by a program write down X in binary from the description"*. This minimal description is practically almost the same as Kolmogorov complexity. Next, consider the program:

```
For i=1,2,...
        If (K(i) >= 1000) output i
```

Here we have a really short program that outputs the number i and thus nicely captures the paradox. It should be noted that it requires a subroutine that computes $K(i)$ to complete the paradox. However, there is no program that computes Kolmogorov complexity. Therefore, there is no paradox.

Theorem 3. *$K(x)$ is not computable.*

Proof. If $K(x)$ was computable by a program of size c then the program:

```
For i=1,2,...
        If (K(i) >= 2c) output i
```

would be of size $\sim c$ and output a string with $K(x) \geq 2c$. By the definition of Kolmogorov complexity we reach a contradiction. □

Next we turn to the notion of entropy.

Definition 2. *Let X be a random variable that takes m values, where the probability of the i'th value is p_i. The entropy of X defined by:*

$$H(X) = \sum_{i=0}^{m-1} -p_i \log p_i \qquad \text{(log of base 2)}$$

Theorem 4. *The expected number of bits needed to code an observation of X is at least $H(X)$.*

As a converse we can mention that the Huffman code of X uses on average at most $H(x) + 1$ bits to code an observation of X. The problem of coding might seem to be solved by this tight bound but in fact it is not. Consider the case whan X is an English text of length 1000. Then the number of possible outcomes m is about 26^{1000}. Also, the values p_i's are unknown (we need to know the probability of all the possible texts). If we code text character by character then $m = 26$ and we can compute p_i's and a good tree. However, the actual compression is much worse as characters are dependent.

Let X and Y be independent, then it's intuitively clear that we might as well code X and Y separately. This corresponds to the mathematical fact that for independent variables entropy is additive, i.e. $H(X,Y) = H(X) + H(Y)$. However, if we have a dependency we should be able to do better and in fact for dependent variables we have $H(X,Y) < H(X) + H(Y)$. That is, we can somehow code X, Y together and use less bits.

3 Lempel-Ziv compression algorithms

Lets consider practical compression of real text, or strings in general. We cover two algorithms, both by Lempel and Ziv; LZ 77 and LZ 78, published in 1977 and 1978, respectively.

3.1 LZ 77: Sliding window

A token in LZ 77 consists of (i_1, i_2, c), where i_1 and i_2 are integers and c is a character. The tokens are interpreted as: *"Go back i_1 positions in the text and copy i_2 positions from there. Add character c.* It operates on the assumption that what is currently being seen in the text probably has been seen before. As an example consider the string "*aaa...a*" with n a's, i.e., with the following tokens:

1. $(0, 0,$'a'$)$: No possibility to copy as there's nothing to copy for the first position (we need to start with something). Copy nothing. Add 'a'.

2. $(1, n-2,$'a'$)$: Go back 1 step. Copy $n-2$ size string. Add the last 'a'.

All of the a's are now coded. Of course, LZ 77 doesn't perform as nice as this in general, however, it still does well in practice. The compression step is easy, i.e., set up a hash table to find matches. Decompressing is very easy, i.e., use a buffer for the text, read a token and copy.

A problem with this algorithm is that i_1, i_2 needs to be coded, and specifically, we need to know where i_1 stops and i_2 starts (and where i_2 stops, as we also need to know where c starts). In theory, we could use some prefix-free encoding of integers. In practice, one solution is to use fixed sizes for i_1, i_2 say $i_1 \leq 2047$ (11 bits), $i_2 \leq 7$ (3 bits).

3.2 LZ 78: Maintain a dictionary

A token in LZ78 consists of (i, c), where i is an integer and c a character. The algorithm operates by finding matchings of substrings which is stored in a dictionary. The dictionary can either be initialized as empty or containing entries for all single symbols. When encoding a string, find the largest matching substring. Copy the substring and concatenate the character c to form a new string. Store this new string at the next empty position in the dictionary.

Some practical considerations are how to code i and how to decide the maximal size of the dictionary. Also, one must consider what to do when the dictionary gets full. Two options are to either erase it (and start from scratch), or to keep working with a static dictionary.

Theorem 5. *Let X be a random variable. Suppose we code n independent observations of X then, LZ 77 and LZ 78 in unbounded form uses $(1 + o(1)) \cdot n \cdot H(x)$ bits, where $o(1) \rightarrow 0$ when $n \rightarrow \infty$.*

Lets consider a string x where LZ 77 and LZ 78 do poorly but where $K(x)$ is small. One example could be the output of a Pseudo-Random Generator. This by definition has small $K(x)$, however, if it's a good generator it won't have any repetitions and therefore LZ 77 and LZ 78 will perform badly.