12 Huffman Coding

Huffman coding is a method to code a sequence of data items with the minimum number of bit necessary. To see how this works, let us look at an example. Consider the text "`abbbaabaaabaaa`". It can be coded using the ASCII code (7 bit per character) and requires then $14 \cdot 7 = 98$ bit. Alternatively, we can use ISO-8859 Code (8 bit per character) and will need 112 bit. If we care for the minimum amount of bit, we can use a 0-bit for an "`a`" and a 1-bit for a "`b`" and get by with only 14 bit.

That was easy. But how about coding "`abcdaabaaabaaa`"? Now, we can no longer use one bit per character, and we might consider using two bit per character.

Code	Data
00	`a`
01	`b`
10	`c`
11	`d`

Tab. 80: Encoding 1

Code	Data
0	`a`
10	`b`
110	`c`
111	`d`

Tab. 81: Encoding 2

Code	Data
0	`a`
1	`b`
01	`c`
11	`d`

Tab. 82: Encoding 3

Encoding 1 brings us to $14 \cdot 2 = 28$ bit. This, however is not optimal. If we choose encoding 2, we need only 21 bit (9 bit for 9 "`a`", 6 bit for 3 "`b`", 3 bit for 1 "`c`", and 3 bit for 1 "`d`"). This is a saving of 25 % compared to encoding 1.

The principle is simple. For codes that occur more often, we use short bit sequences and for codes that occur infrequently, we use longer bit sequences. The extra spending for a few long codes is easily recovered with many short codes.

But may be there are still better codings. For instance, we could use encoding 3. This would require a total of only $9 \cdot 1 + 3 \cdot 1 + 1 \cdot 2 + 1 \cdot 2 = 16$ bit. Unfortunately, it won't work. We can encode the text to yield "`0101110010001000`", and it is only 16 bit long, but we can no longer decode it unambiguously. We could take the leading "`01`" either for an "`ab`" or for a "`c`". Similar for the next two bit. And how should we know that in the first case it is "`ab`" and in the second "`c`"?

Let us return to encoding 2, which yields "`01011011100100010000`", and study the decoding process in detail. We start reading a 0-bit. How can we know that this is an "`a`" and not the start of a longer code for a different character? Simple: all the other codes start with a 1. So it's an "`a`". We read the second bit: 1. Obviously,

this is not an "a". It is not a code for any character. So we read on and get an other 0-bit. This is a "b". How do we know? The reason is as before: There is no other code that starts with "10". In summary: to make unambiguous decoding feasible, no code must be the prefix of an other code.

Codes that have this prefix property can be depicted as binary decision trees. Encoding 2 is shown in Fig. 83 as a binary tree.

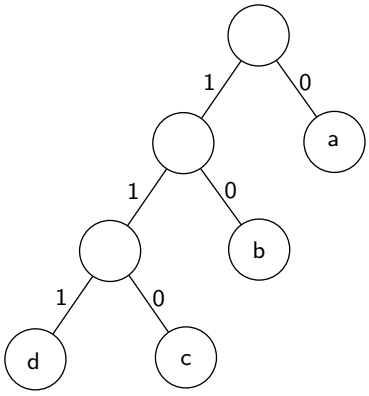

Fig. 83: Huffman tree for encoding 2

Each code describes a path from the root of the tree to a leaf node. A 0-bit selects the branch to the right, a 1-bit selects a branch to the left. Once we arrive at a leaf node, we have found the correct decoded character.

Back to the question of an optimal encoding that allows unambiguous decoding. This is the problem for which Huffman coding is the answer[15]. We omit the details here and recommend reading a good book on algorithms (for example [33] pages 351–357) instead.

In our case, the encodings are given by the standard and we just have to use them. We better focus our curiosity on the problem of optimal decoding of a given encoding.

Naturally, it is possible to choose a representation of the binary Huffman tree and decode a bit stream by traversing the tree. Each time we encounter a leaf node, we emit the corresponding character and start over from the root of the tree. The runtime of this algorithm is proportional to the number of bit in the bit stream, and this is not bad. Since the most frequent characters will have short codes, they can be decoded very fast.

An alternative is decoding with a lookup table. The index, we want to use is the Huffman code, and the result is the corresponding character. Our Huffman codes are, however, of variable length, and we do not even know the exact length before we have decoded them. We solve this problem, by taking a bit string of maximum code length from the input and use it as an index. For a short code this means that there must be many different entries in our table, one for every possible extension of the code up to the maximum code length.

For encoding 2, the required table is shown in Tab. 84. Observe, that the character "a" requires four entries in the table. If the index starts with a 0-bit, it must lead to the character "a", whatever the next two bit of the index happen to be.

The bit stream "01011011100100001000", which we used above as an example, starts with "010". We use it as an index to find the first character "a" and the length of the code: 1 bit. We advance the input by one bit and take the next three bit: "101". Using these as an index, we obtain the "b" and its code length: 2.

The use of the lookup table will yield a faster algorithm but requires more memory.

Index	Index (binary)	Code	Bit required
0	000	a	1
1	001	a	1
2	010	a	1
3	011	a	1
4	100	b	2
5	101	b	2
6	110	c	3
7	111	d	3

Tab. 84: Lookup table for encoding 2

The runtime of the algorithm is now proportional to the number of decoded characters, and all the characters are decoded with the same speed. The space requirements for the tables will grow exponentially with the maximum code length. The maximum code length used in the encodings of the standard is 19, and this would require a table with approximately a half a million entries—too much space wasted for a few very long codes that are not used very often.

Hence, we combine the two approaches. We restrict the lookup tables to a fixed maximum number of bit, `HWIDTH`, to be used for the index. If this number of bit is enough to decode the character, that is if its code length is less or equal to `HWIDTH`, the table will immediately reveal the character and its code length. If the code length is greater than `HWIDTH`, the table will provide a pointer to a new table, called an extension table, and the number of bit required for the index of the extension table. In effect, we replace the binary tree by an n-way tree, where $n = 2^{\texttt{HWIDTH}}$ is the table size. It provides access to the most common codes with a single table lookup and, for longer codes, has a runtime proportional to the code length.

HWIDTH	Table Entries
1	2726
2	2172
3	2084
4	2176
5	2398
6	2950
7	3746
8	5486

Tab. 85: Total memory usage

HWIDTH	Max. Lookups	Runtime
1	19	100 %
2	10	75.4 %
3	7	61.7 %
4	5	56.5 %
5	4	53.6 %
6	4	51.2 %
7	3	50.5 %
8	3	49.2 %

Tab. 86: Time requirements for encoding 15

The choice of `HWIDTH` depends on the time/space tradeoff we are willing to make.