Coding Trees in the Deflate format

Christoph-Simon Senjak

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

PUMA/AlgoSyn Workshop 26. März 2013

Outline

- Deflate is probably the most widespread compression format
- We use it as case study for low-level verification
- Deflates way to store encoding trees (huffman trees) is a lot more complicated than expected

Deflate - Basics

- Specified in RFC 1951. (RFC 1952 specifies the GZIP file format, and RFC 1950 specifies the ZLIB file format. Both are often confused with Deflate).

Deflate - Basics

- Specified in RFC 1951. (RFC 1952 specifies the GZIP file format, and RFC 1950 specifies the ZLIB file format. Both are often confused with Deflate).
- Can make use of Huffman-Codings and (extended) Run-length encoding.

Deflate - Basics

- Specified in RFC 1951. (RFC 1952 specifies the GZIP file format, and RFC 1950 specifies the ZLIB file format. Both are often confused with Deflate).
- Can make use of Huffman-Codings and (extended) Run-length encoding.
- Does not require any specific compression algorithm, even though some are recommended.

Deflate - Basics

- Specified in RFC 1951. (RFC 1952 specifies the GZIP file format, and RFC 1950 specifies the ZLIB file format. Both are often confused with Deflate).
- Can make use of Huffman-Codings and (extended) Run-length encoding.
- Does not require any specific compression algorithm, even though some are recommended.
- Widely used: HTTP, ZIP/RAR, PNG, SSH

```
$ apt-cache rdepends zlib1g | wc -l
1418
```

Deflate - Overview

An informal illustration of the format:

```
Deflate              ::=  ('0' Block)* '1' Block (0|1)*
Block                ::=  '00' UncompressedBlock |
                          '01' DynamicallyCompBl |
                          '10' StaticallyCompBl
UncompressedBlock    ::=  length ~length bytes
StaticallyCompBl     ::=  CompBl(standard coding)
DynamicallyCompBl    ::=  header coding CompBl(coding)
CompBl(c)            ::=  [^256]* 256
```

Deflate - Overview

An informal illustration of the format:

```
Deflate              ::=  ('0' Block)* '1' Block (0|1)*
Block                ::=  '00' UncompressedBlock |
                          '01' DynamicallyCompBl |
                          '10' StaticallyCompBl
UncompressedBlock    ::=  length ~length bytes
StaticallyCompBl     ::=  CompBl(standard coding)
DynamicallyCompBl    ::=  header coding CompBl(coding)
CompBl(c)            ::=  [^256]* 256
```

We are interested in the "coding" part.

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent. (RFC 1951, 3.2.2)

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent. (RFC 1951, 3.2.2)
4. If there is a lexicographically smaller code c of the same length for some code $D(a)$ in the coding, then it is prefixed by some image of the coding D:
   $\forall_{a,c}.c \sqsubseteq D(a) \rightarrow \operatorname{len} c = \operatorname{len} D(a) \rightarrow \exists_b.D(b) \neq [] \wedge D(b) \preccurlyeq c$

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent. (RFC 1951, 3.2.2)
4. If there is a lexicographically smaller code c of the same length for some code $D(a)$ in the coding, then it is prefixed by some image of the coding D:
   $\forall_{a,c}.c \sqsubseteq D(a) \rightarrow \operatorname{len} c = \operatorname{len} D(a) \rightarrow \exists_b.D(b) \neq [] \wedge D(b) \preccurlyeq c$

Lemma: If c is a deflate coding, then so are $c_i x = \begin{cases} y & \text{for } cx = i :: y \\ [] & \text{otherwise} \end{cases}$ for $i \in \{0, 1\}$

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent. (RFC 1951, 3.2.2)
4. If there is a lexicographically smaller code c of the same length for some code $D(a)$ in the coding, then it is prefixed by some image of the coding D:
   $\forall_{a,c}.c \sqsubseteq D(a) \rightarrow \operatorname{len} c = \operatorname{len} D(a) \rightarrow \exists_b.D(b) \neq [] \wedge D(b) \preccurlyeq c$

Lemma: If c is a deflate coding, then so are $c_i x = \begin{cases} y & \text{for } cx = i :: y \\ [] & \text{otherwise} \end{cases}$ for $i \in \{0, 1\}$ $\Rightarrow$ Coding trees as internal representation possible, but too expensive for storage.

Deflate Codings

1. Must be **prefix-free**, that is, no code prefixes another code, so they form a tree.
2. The shorter codes must lexicographically precede longer codes. (RFC 1951, 3.2.2)
3. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent. (RFC 1951, 3.2.2)
4. If there is a lexicographically smaller code c of the same length for some code $D(a)$ in the coding, then it is prefixed by some image of the coding D:
   $\forall_{a,c}.c \sqsubseteq D(a) \rightarrow \operatorname{len} c = \operatorname{len} D(a) \rightarrow \exists_b.D(b) \neq [] \wedge D(b) \preccurlyeq c$

Lemma: If c is a deflate coding, then so are $c_i x = \begin{cases} y & \text{for } cx = i :: y \\ [] & \text{otherwise} \end{cases}$ for $i \in \{0, 1\}$ $\Rightarrow$ Coding trees as internal representation possible, but too expensive for storage. 4. is complicated but crucial to find a simpler representation.

Deflate Coding Sequences

- A **deflate sequence** is a sequence $(a_i)_i$ of code-lengts or 0 that satisfies **Kraft's inequality** $\sum_{i, c_i \neq 0} 2^{-a_i} \leq 1$.

Deflate Coding Sequences

- A **deflate sequence** is a sequence $(a_i)_i$ of code-lengts or 0 that satisfies **Kraft's inequality** $\sum_{i, c_i \neq 0} 2^{-a_i} \leq 1$.
- **Main Theorem:** The type of lists that satisfy Kraft's inequality is isomorphic to the type of Deflate codings.

Deflate Coding Sequences

- A **deflate sequence** is a sequence $(a_i)_i$ of code-lengts or 0 that satisfies **Kraft's inequality** $\sum_{i, c_i \neq 0} 2^{-a_i} \leq 1$.
- **Main Theorem:** The type of lists that satisfy Kraft's inequality is isomorphic to the type of Deflate codings.

$\Rightarrow$ Saving a sequence of lengths is not expensive, and that is the way it is done in Deflate, and is sufficient, as this theorem shows.

Deflate Coding Sequences

- A **deflate sequence** is a sequence $(a_i)_i$ of code-lengts or 0 that satisfies **Kraft's inequality** $\sum_{i, c_i \neq 0} 2^{-a_i} \leq 1$.
- **Main Theorem:** The type of lists that satisfy Kraft's inequality is isomorphic to the type of Deflate codings.

$\Rightarrow$ Saving a sequence of lengths is not expensive, and that is the way it is done in Deflate, and is sufficient, as this theorem shows.

- We made a detailled informal proof and now start to formalize it in Coq.

Conclusion

Conclusion

- The algorithm as such can be implemented very efficiently.

Conclusion

- The algorithm as such can be implemented very efficiently.
- Verification is, however, very complicated.

Conclusion

- The algorithm as such can be implemented very efficiently.
- Verification is, however, very complicated.

$\Rightarrow$ Even more useful to have a verified reference implementation.