Polynomial Time Algorithms for Constructing Optimal AIFV Codes
Version of January 27, 2019

Mordecai Golin and Elfarouk Harb
Hong Kong UST

To appear in DCC’19
Huffman encoding is an “optimal” lossless compression algorithm. Optimality implicitly uses two unstated conditions:
(i) only one encoding (tree node) per source letter and
(ii) encoding is instantaneous.
   i.e., can decode a letter as soon as its final bit is seen.

Relaxing those two conditions permits constructing Almost Instantaneous Fixed to Variable (AIFV) code that beat Huffman.
Construction techniques are complicated:
using ellipsoid methods to find finite-state Markov Chains that have “optimal” steady state distributions.

Lots of open problems remaining.
Finding better AIFV codes.
Finding faster algorithms.
Finding strongly polynomial algorithms.
Outline

- **Introduction**
- **AIFV-2 codes: cost and algorithm**
- **A Geometric Interpretation of the old algorithm**
  - A New Binary Search Algorithm
  - An Ellipsoid Algorithm
- **Extensions to AIFV-\(k\) codes (skip)**
- **Summing up and open questions**
● Huffman coding is a lossless data compression algorithm.
Huffman coding is a lossless data compression algorithm.

Let $\mathcal{X}$ be a finite alphabet of size $n$ (e.g. $\mathcal{X} = \{a, b, c, d\}$)
• Huffman coding is a lossless data compression algorithm.

• Let $\mathcal{X}$ be finite alphabet of size $n$ (e.g. $\mathcal{X} = \{a, b, c, d\}$)

• $\forall x \in \mathcal{X}$, let $p_x = p_X(x)$ be probability of source letter $x$ occurring, e.g., $p_a = 0.5, p_b = 0.3, p_c = 0.15, p_d = 0.05$. 
• Huffman coding is a lossless data compression algorithm.

• Let $\mathcal{X}$ be finite alphabet of size $n$ (e.g. $\mathcal{X} = \{a, b, c, d\}$)

• $\forall x \in \mathcal{X}$, let $p_x = p_{\mathcal{X}}(x)$ be probability of source letter $x$ occurring, e.g., $p_a = 0.5, p_b = 0.3, p_c = 0.15, p_d = 0.05$.

• $c \in \{0, 1\}^*$ is a codeword, e.g., $c = 0111$. $|c|$ denotes the length of the codeword, e.g., $|0111| = 4$. 
Let \( X \) be finite alphabet of size \( n \) (e.g. \( X = \{a, b, c, d\} \))

\[ \forall x \in X, \ p_x = p_X(x) \text{ be probability of source letter } x \]

\[ p_a = 0.5, \ p_b = 0.3, \ p_c = 0.15, \ p_d = 0.05. \]

\( c \in \{0, 1\}^* \) is a codeword, e.g., \( c = 0111 \). 
\[ |c| \text{ denotes the length of the codeword, e.g., } |0111| = 4. \]

A code is a mapping \( C \) of source letters to codewords, e.g. \( C(a) = 01, \ C(b) = 0010, \ C(c) = 1001, \ C(d) = 001. \)
• Average code length of code $C$ over source $\mathcal{X}$ is

$$L(C) = \sum_{x \in \mathcal{X}} |C(x)|p_x$$
• Average code length of code $C$ over source $\mathcal{X}$ is

$$L(C) = \sum_{x \in \mathcal{X}} |C(x)| p_x$$

• Example: if $\mathcal{X} = \{a, b, c, d\}$

$$p_a = 0.5, p_b = 0.3, p_c = 0.15, p_d = 0.05$$

$$C(a) = 01, C(b) = 001, C(c) = 0001, C(d) = 0000$$
• Average code length of code $C$ over source $\mathcal{X}$ is

$$L(C) = \sum_{x \in \mathcal{X}} |C(x)|p_x$$

• Example: if $\mathcal{X} = \{a, b, c, d\}$

$$p_a = 0.5, p_b = 0.3, p_c = 0.15, p_d = 0.05$$

$$C(a) = 01, C(b) = 001, C(c) = 0001, C(d) = 0000$$

• ⇒ the average code length is

$$L(C) = |C(a)|p_a + |C(b)|p_b + |C(c)|p_c + |C(d)|p_d$$

$$= 2 \times 0.5 + 3 \times 0.3 + 4 \times 0.15 + 4 \times 0.05 = 2.7$$
• Given Source alphabet $\mathcal{X}$ and its probability distribution, find prefix-free code $C$ that minimizes average code length $L(C')$. 
• Given Source alphabet $\mathcal{X}$ and its probability distribution, find prefix-free code $C$ that minimizes average code length $L(C)$.

• Huffman Coding does this.

![Huffman Coding Example](image)
Given Source alphabet $\mathcal{X}$ and its probability distribution, find prefix-free code $C$ that minimizes average code length $L(C)$.

Huffman Coding does this.

Each leaf in tree corresponds to source letter $x \in \mathcal{X}$

- $C(a) = 0$
- $C(b) = 10$
- $C(c) = 110$
- $C(d) = 111$
Given a Huffman Code, recall how to encode/decode.
Given a Huffman Code, recall how to encode/decode.

How to encode \textit{daba}?
- Concatenatate codewords for \textit{d, a, b, a}
Given a Huffman Code, recall how to encode/decode.

How to encode $daba$ ?

- Concatenate codewords for $d, a, b, a$
  - $C(d) = 111$
  - $C(a) = 0$
  - $C(b) = 10$
Given a Huffman Code, recall how to encode/decode.

How to encode \textit{daba}?

- Concatenate codewords for \textit{d, a, b, a}
  - \( C(d) = 111 \)
  - \( C(a) = 0 \)
  - \( C(b) = 10 \)

\textit{daba} is encoded as \texttt{1110100}
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110 ?
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?
Trace the code word bit-by-bit until reaching a leaf. Then restart.

111110110
Given a Huffman Code, recall how to encode/decode.

How to decode `111110110`?

Trace the code word bit-by-bit until reaching a leaf. Then restart.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?
Trace the code word bit-by-bit until reaching a leaf. Then restart.

111110110
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?

Trace the code word bit-by-bit until reaching a leaf. Then restart.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?

Trace the code word bit-by-bit until reaching a leaf. Then restart.

Stop! Reached leaf corresponding to d, so we decode as d.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?
Trace the code word bit-by-bit until reaching a leaf. Then restart.

Stop! Reached leaf corresponding to d, so we decode as d.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?
Trace the code word bit-by-bit until reaching a leaf. Then restart.

Stop! Reached leaf corresponding to c so decode as c.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?

Trace the code word bit-by-bit until reaching a leaf. Then restart.

Stop! Reached leaf corresponding to c so decode as c.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?
Trace the code word bit-by-bit until reaching a leaf. Then restart.

Similarly, next 110 is also decoded as $c$.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?

Trace the code word bit-by-bit until reaching a leaf. Then restart.

Similarly, next 110 is also decoded as c.

Similarly, next 110 is also decoded as c.
Given a Huffman Code, recall how to encode/decode.

How to decode 111110110?

Trace the code word bit-by-bit until reaching a leaf. Then restart.

Similarly, next 110 is also decoded as c.

Hence, 111110110 is decoded as dcc
• Huffman Coding optimality proof uses two implicit assumptions.
  • The decoding procedure is **instantaneous**
  • Any code can be represented as a **single** code tree.
• Huffman Coding optimality proof uses two implicit assumptions.
  • The decoding procedure is **instantaneous**
  • Any code can be represented as a **single** code tree.

• **Instantaneous** means that immediately after reading the last bit in a codeword, the source character is known.
No decoding delay is allowed once a bit is read.
Huffman Coding optimality proof uses two implicit assumptions.

- The decoding procedure is **instantaneous**
- Any code can be represented as a **single** code tree.

**Instantaneous** means that immediately after reading the last bit in a codeword, the source character is known.

No decoding delay is allowed once a bit is read.

Assumptions are a bit restrictive.

Can Huffman Coding compression rate be beaten if the assumptions are relaxed?
Huffman Coding optimality proof uses two implicit assumptions.

- The decoding procedure is **instantaneous**
- Any code can be represented as a **single** code tree.

**Instantaneous** means that immediately after reading the last bit in a codeword, the source character is known.

No decoding delay is allowed once a bit is read.

- Assumptions are a bit restrictive.

Can Huffman Coding compression rate be beaten if the assumptions are relaxed?

- Yes !
• An *Almost Instantaneous Code* might require a **bounded** decoding delay.
• An *Almost Instantaneous Code* might require a *bounded* decoding delay.

• An AIFV-2 Code is an *Almost Instantaneous Code* that has a decoding delay at most 2, i.e., might need to read 2 bits after codeword ends before recognizing codeword.
• An *Almost Instantaneous Code* might require a **bounded** decoding delay.

• An **AIFV-2 Code** is an *Almost Instantaneous Code* that has a decoding delay at most 2, i.e., might need to read 2 bits *after* codeword ends before recognizing codeword.

• Each **AIFV-2 code** is represented by two code trees $T_0, T_1$. Each $x \in \mathcal{X}$ is represented by **two** codewords: one in each tree.
• An *Almost Instantaneous Code* might require a **bounded** decoding delay.

• An **AIFV-2 Code** is an *Almost Instantaneous Code* that has a decoding delay at most 2, i.e., might need to read 2 bits *after* codeword ends before recognizing codeword.

• Each AIFV-2 code is represented by two code trees $T_0, T_1$. Each $x \in X$ is represented by **two** codewords: one in each tree.

![Diagram of two code trees $T_0$ and $T_1$ representing the AIFV-2 code. Each leaf node is labeled with a symbol from the set $\{a, b, c, d\}$. The tree $T_0$ has nodes labeled 0, 1, 0, 1, 0, 1, while $T_1$ has nodes labeled 0, 1, 0, 1, 0, 1.]}
• An *Almost Instantaneous Code* might require a **bounded** decoding delay.

• An **AIFV-2 Code** is an *Almost Instantaneous Code* that has a decoding delay at most 2, i.e., might need to read 2 bits after codeword ends before recognizing codeword.

• Each AIFV-2 code is represented by two code trees $T_0, T_1$. Each $x \in \mathcal{X}$ is represented by **two** codewords: one in each tree.

$$C_0(a) = 0, \quad C_1(a) = 01$$
• An *Almost Instantaneous Code* might require a **bounded** decoding delay.

• An AIFV-2 Code is an *Almost Instantaneous Code* that has a decoding delay at most 2, i.e., might need to read 2 bits after codeword ends before recognizing codeword.

• Each AIFV-2 code is represented by two code trees $T_0, T_1$. Each $x \in \mathcal{X}$ is represented by **two** codewords: one in each tree.

• $C_0(a) = 0$, $C_1(a) = 01$

• $C_0(b) = 10$, $C_1(b) = 10$

• $C_0(c) = 11$, $C_1(c) = 11$

• $C_0(d) = 1000$, $C_1(d) = 1100$
Definition of AIFV-2 Code $T_0, T_1$
Defintion of AIFV-2 Code $T_0$, $T_1$

Root of $T_1$ is complete.
0 child of root only has a 1 child.
Incomplete internal nodes (with exception above) have only a 0 child.
Definition of AIFV-2 Code $T_0, T_1$

Root of $T_1$ is complete.
0 child of root only has a 1 child.
Incomplete internal nodes (with exception above) have only a 0 child.
Incomplete nodes are labelled as either master or slave nodes.

Master nodes are incomplete nodes with incomplete children.
Defintion of AIFV-2 Code $T_0, T_1$

Codewords are leaves and master nodes.
Slave nodes and complete internal nodes are not codewords.

Root of $T_1$ is complete.
0 child of root only has a 1 child.
Incomplete internal nodes (with exception above) have only a 0 child.
Incomplete nodes are labelled as either master or slave nodes.

Master nodes are incomplete nodes with incomplete children.
Encoding/Decoding with AIFV-2 Codes $T_0, T_1$

Encoding $S = s_1, s_2, \ldots s_k \in \mathcal{X}^k$
Encoding/Decoding with AIFV-2 Codes $T_0, T_1$

Encoding $S = s_1, s_2, \ldots s_k \in \mathcal{X}_k$

*Master nodes* are internal node codewords.
Encoding/Decoding with AIFV-2 Codes $T_0, T_1$

Encoding $S = s_1, s_2, \ldots s_k \in \mathcal{X}^k$

Master nodes are internal node codewords.

Encode $s_1$ with tree $T_0$

For $i = 2$ to $k$

if $s_{i-1}$ was encoded using a master node

encode $s_i$ with tree $T_1$

else:

encode $s_i$ with tree $T_0$
Example: Encoding \textit{dabcab}

\begin{itemize}
  \item \textbf{T}0
  \begin{itemize}
    \item 0
    \begin{itemize}
      \item \textit{a}
      \begin{itemize}
        \item 1
        \begin{itemize}
          \item \textit{b}
          \begin{itemize}
            \item 0
            \begin{itemize}
              \item \textit{d}
              \begin{itemize}
                \item 0
              \end{itemize}
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
  \item 1
  \begin{itemize}
    \item \textit{c}
    \begin{itemize}
      \item 0
      \begin{itemize}
        \item \textit{a}
        \begin{itemize}
          \item 1
          \begin{itemize}
            \item \textit{b}
            \begin{itemize}
              \item 0
              \begin{itemize}
                \item \textit{c}
                \begin{itemize}
                  \item 0
                \end{itemize}
              \end{itemize}
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}

\textit{dabcab}
Example: Encoding $dabcab$

Start in $T_0$.
Encode $d$ as $C_0(d) = 1000$
Example: Encoding $dabcab$

Start in $T_0$.
Encode $d$ as $C_0(d) = 1000$
Example: Encoding $dabcab$

Start in $T_0$.
Encode $d$ as $C_0(d) = 1000$
$d$ is not master $\Rightarrow$ stay in $T_0$
Example: Encoding $dabcab$

Start in $T_0$. Encode $a$ as $C_0(a) = 0$.

$a$ is not master $\Rightarrow$ stay in $T_0$.  

$dabcab$

1000 0

$d$  $a$
Example: Encoding $dabcab$

Start in $T_0$.  
Encode $b$ as $C_0(b) = 10$ 
$b$ is a master $\Rightarrow$ switch to $T_1$
Example: Encoding $dabcab$

Start in $T_1$.
Encode $c$ as $C_1(c) = 11$
$c$ is a master $\Rightarrow$ stay in $T_1$
Example: Encoding $dabcab$

Start in $T_1$.  
Encode $a$ as $C_1(a) = 01$  
$a$ is not a master $\Rightarrow$ switch to $T_0$

1000 0 10 11 01  
d a b c a
Example: Encoding \textit{dabcab}

Start in $T_0$. Encode $b$ as $C_0(b) = 10$.
Example: Encoding \textit{dabcab}
The Decoding Procedure

Start at $T_0$ and trace codeword through tree.
The Decoding Procedure

Start at $T_0$ and trace codeword through tree.

If a leaf is reached, decode using that word.

If decoding is “blocked” due to missing ”1” edge, go back to last master seen and use it as decoded letter.
Start at $T_0$ and trace codeword through tree.

If a leaf is reached, decode using that word.

If decoding is “blocked” due to missing ”1” edge, go back to last master seen and use it as decoded letter.

Similar to encoding, if last symbol decoded used master, use $T_1$ for next symbol; otherwise use $T_0$
Example: Decoding 1000010110110

$T_0$

$T_1$

1000010110110
Example: Decoding 1000010110110

T0

1000010110110

T1

1000010110110

T1

T0
Example: Decoding 1000010110110

$T_0$

$T_1$

1000010110110
Example: Decoding $1000010110110$
Example: Decoding 1000010110110

\[ T_0 \]

\[ T_1 \]

1000010110110
Example: Decoding 1000010110110

Decode $d$. Since $d$ is not master, remain in $T_0$. 

```
1000010110110
```
Example: Decoding 1000010110110
Example: Decoding 1000010110110

Decode a.
Since a is not master, remain in $T_0$
Example: Decoding 1000010110110
Example: Decoding 1000010110110
Example: Decoding 1000010110110

Trace is blocked.
Codeword has 1, but code tree only has 0 edge. Must use master node b.
Example: Decoding 1000010110110

Trace is blocked.
Codeword has 1, but code tree only has 0 edge.
Must use master node $b$. 
Example: Decoding $1000010110110$

Since $b$ is a master node, switch to $T_1$. 
Example: Decoding 1000010110110110

\[ \begin{array}{c}
T_0 \\
0 & 1 \\
a & b \\
0 & 1 \\
b & c \\
0 & 1 \\
c & d \\
0 & 0 \\
d & d \\
\end{array} \]

\[ \begin{array}{c}
T_1 \\
0 & 1 \\
a & b \\
1 & 0 \\
c & c \\
1 & 1 \\
d & d \\
0 & 0 \\
d & d \\
\end{array} \]

\[ \begin{array}{c|c|c|c}
\text{d} & \text{a} & \text{b} \\
1000 & 01 & 101110101 \\
\end{array} \]
Example: Decoding 1000010110110
Example: Decoding 1000010110110
Example: Decoding 1000010110110

Trace is blocked again. Code word has 1 but tree only has 0 edge. Must use master node \( c \).
Example: Decoding 1000010110110

Trace is blocked again.
Code word has 1 but tree only has 0 edge.
Must use master node c.
Example: Decoding 1000010110110

Since $c$ is a master node, remain in $T_1$. 
Example: Decoding 1000010110110

\[
\begin{array}{cccc}
T_0 & & T_1 & \\
0 & 1 & 0 & 1 \\
a & b & c & a \\
0 & 1 & 0 & 1 \\
b & c & d & d \\
0 & 0 & 0 & 0 \\
d & d & d & d \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
d & a & b & c \\
1000010110110 \\
\end{array}
\]
Example: Decoding 1000010110110
Example: Decoding 1000010110110

Decode a.
Since a is not master, switch to $T_0$.
Example: Decoding 1000010110110

\[ \begin{array}{cccc}
& T_0 & & T_1 \\
\text{a} & 0 & 1 & 1 \text{a} \\
\text{b} & 0 & 1 & 0 \text{b} \\
\text{c} & 0 & 1 & 0 \text{c} \\
\text{d} & 0 & 1 & 0 \text{d} \\
\end{array} \]
Example: Decoding 1000010110110

\[ T_0 \]

\[ T_1 \]

\[
\begin{array}{llllll}
  & d & a & b & c & a \\
1000 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
\]
Example: Decoding 1000010110110
Example: Decoding 1000010110110

The final decoded word is \textit{dabcab}
• Optimal AIFV-2 Codes compress at least as well as Huffman coding. There are examples (such as the last example, calculation later) that can be shown to beat Huffman compression.

• Allowing a decoding delay of 2 bits, and 2 trees permits improving the compression.
• Optimal AIFV-2 Codes compress at least as well as Huffman coding. There are examples (such as the last example, calculation later) that can be shown to beat Huffman compression.

• Allowing a decoding delay of 2 bits, and 2 trees permits improving the compression.

• Constructing Optimal Huffman Codes is $O(n \log n)$, or $O(n)$ if the probabilities are sorted.

• Constructing Optimal AIFV-2 codes is much more difficult. State of the art had no polynomial algorithm.
References and Extensions

General AIFV References

(1) H. Yamamoto and X. Wei, “Almost instantaneous FV codes,” 2013 IEEE ISIT


AIFV-$m$ Codes (a generalization to $m$ coding trees)

(4) H. Yamamoto and K. Iwata, “An iterative algorithm to construct optimal binary AIFV-$m$ codes,” IEEE ITW’17

(5) K. Iwata and H. Yamamoto, “A dynamic programming algorithm to construct optimal code trees of AIFV codes,” ISITA’16,
Outline

- Introduction
- AIFV-2 codes: cost and algorithm
- A Geometric Interpretation of the old algorithm
  - A New Binary Search Algorithm
  - An Ellipsoid Algorithm
- Extensions to AIFV-\(k\) codes (skip)
- Summing up and open questions
Calculating average code length $L_{AIFV}(T_0, T_1)$

∀$x \in \mathcal{X}$, let $c_s(x)$ be the code word representing $x$ in $T_s$.

The **average length** of individual code tree $T_s$ is

$$L(T_s) = \sum_{x \in \mathcal{X}} |c_s(x)| p_x$$
Calculating average code length $L_{AIFV}(T_0, T_1)$

Fix $T_0, T_1$.
Consider randomly generated string $S = s_1, s_2, \ldots, \in \mathcal{X}^*$.

The tree used to encode $s_i$ is modelled by a two state ergodic Markov Chain.
Fix $T_0, T_1$.
Consider randomly generated string $S = s_1, s_2, \ldots, \in \mathcal{X}^*$.

The tree used to encode $s_i$ is modelled by a two state ergodic Markov Chain.

Let $q_0(T_1)$ be sum of leaf weights in $T_1$; $q_1(T_0)$ the sum of master weights in $T_0$. 

Calculating average code length $L_{AIFV}(T_0, T_1)$
Calculating average code length \( L_{AIFV}(T_0, T_1) \)

Fix \( T_0, T_1 \).

Consider randomly generated string \( S = s_1, s_2, \ldots, \in X^* \).

The tree used to encode \( s_i \) is modelled by a two state ergodic Markov Chain.

Let \( q_0(T_1) \) be sum of leaf weights in \( T_1 \); \( q_1(T_0) \) the sum of master weights in \( T_0 \).

Let \( s, \hat{s} \in \{0, 1\}, s \neq \hat{s} \). Working through the details, the stationary probability of using \( T_s \) is given by

\[
P(s|T_0, T_1) = \frac{q_s(T_{\hat{s}})}{q_0(T_1) + q_1(T_0)}
\]
Calculating average code length $L_{AIFV}(T_0, T_1)$

Fix $T_0, T_1$.
Consider randomly generated string $S = s_1, s_2, \ldots, \in \mathcal{X}^*$.

The tree used to encode $s_i$ is modelled by a two state ergodic Markov Chain.

$$L_{AIFV}(T_0, T_1) = P(0|T_0, T_1)L(T_0) + P(1|T_0, T_1)L(T_1)$$
Calculating average code length $L_{AIFV}(T_0, T_1)$

Fix $T_0, T_1$.
Consider randomly generated string $S = s_1, s_2, \ldots, \in \mathcal{X}^*$.

The tree used to encode $s_i$ is modelled by a two state ergodic Markov Chain.

\[
L_{AIFV}(T_0, T_1) = P(0|T_0, T_1)L(T_0) + P(1|T_0, T_1)L(T_1)
\]

Problem: Find $T_0, T_1$ that minimize $L_{AIFV}(T_0, T_1)$
\[ p_X(a) = 0.5 \quad p_X(b) = 0.25 \]
\[ p_X(c) = 0.2 \quad p_X(d) = 0.05 \]
\[ L(T_0) = 1 \cdot 0.5 + 2 \cdot 0.25 + 2 \cdot 0.2 + 4 \cdot 0.05 = 1.6 \]

\[ L(T_1) = 2 \cdot 0.5 + 2 \cdot 0.25 + 2 \cdot 0.2 + 4 \cdot 0.05 = 2.1 \]

\[ p_X(a) = 0.5 \quad p_X(b) = 0.25 \]
\[ p_X(c) = 0.2 \quad p_X(d) = 0.05 \]
$p_X(a) = 0.5 \quad p_X(b) = 0.25$

$p_X(c) = 0.2 \quad p_X(d) = 0.05$

$L(T_0) = 1 \cdot 0.5 + 2 \cdot 0.25 + 2 \cdot 0.2 + 4 \cdot 0.05 = 1.6$

$L(T_1) = 2 \cdot 0.5 + 2 \cdot 0.25 + 2 \cdot 0.2 + 4 \cdot 0.05 = 2.1$

$q_1(T_0) = 0.25$

$q_0(T_1) = 0.5 + 0.25 + 0.05 = 0.8$
$p_X(a) = 0.5 \quad p_X(b) = 0.25$

$p_X(c) = 0.2 \quad p_X(d) = 0.05$

$L(T_0) = 1 \cdot 0.5 + 2 \cdot 0.25 + 2 \cdot 0.2 + 4 \cdot 0.05 = 1.6$

$L(T_1) = 2 \cdot 0.5 + 2 \cdot 0.25 + 2 \cdot 0.2 + 4 \cdot 0.05 = 2.1$

$q_1(T_0) = 0.25$

$q_0(T_1) = 0.5 + 0.25 + 0.05 = 0.8$

$L_{AIFV}(T_0, T_1) = \frac{1.6 \cdot 0.8 + 2.1 \cdot 0.25}{0.25 + 0.8} < 1.72 < 1.75 = L(\text{Huffman}_a)$
AIFV-2 Construction Algorithm

- Yamamoto et al. proved that this Algorithm constructs optimal AIFV-2 Codes.

Algorithm [Yamamoto et al]

\[
\begin{align*}
m & \leftarrow 0 \\
C(0) & = 2 - \log_2(3) \\
\text{repeat} & \\
& m \leftarrow m + 1 \\
T_0^{(m)} & = \arg\min_{T_0} \{ L(T_0) + C^{(m-1)} q_1(T_0) \} \\
T_1^{(m)} & = \arg\min_{T_1} \{ L(T_1) - C^{(m-1)} q_0(T_1) \} \\
\text{Update cost as} & \\
C^{(m)} & = \frac{L(T_1^{(m)}) - L(T_0^{(m)})}{q_1(T_1^{(m)}) + q_0(T_0^{(m)})} \\
\text{until} & \quad C^{(m)} = C^{(m-1)}
\end{align*}
\]
Yamamoto et al. proved that this Algorithm constructs optimal AIFV-2 Codes.

At each step, algorithm creates two new improved code trees.

**Algorithm [Yamamoto et al]**

\[ m \leftarrow 0 \]
\[ C^{(0)} = 2 - \log_2(3) \]

**repeat**

\[ m \leftarrow m + 1 \]
\[ T_0^{(m)} = \arg\min_{T_0} \{ L(T_0) + C^{(m-1)} q_1(T_0) \} \]
\[ T_1^{(m)} = \arg\min_{T_1} \{ L(T_1) - C^{(m-1)} q_0(T_1) \} \]

Update cost as

\[ C^{(m)} = \frac{L(T_1^{(m)}) - L(T_0^{(m)})}{q_1(T_0^{(m)}) + q_0(T_1^{(m)})} \]

**until** \[ C^{(m)} = C^{(m-1)} \]
Yamamoto et al. proved that this Algorithm constructs optimal AIFV-2 Codes.

At each step, algorithm creates two new improved code trees.

Originally solved using ILP; later replaced by \( O(n^5) \) DP algorithm. Parameterizes trees by “cost” \( C \).

Algorithm [Yamamoto et al]

\[
m \leftarrow 0 \\
C^{(0)} = 2 - \log_2(3) \\
\text{repeat} \\
m \leftarrow m + 1 \\
T_0^{(m)} = \arg\min_{T_0} \{L(T_0) + C^{(m-1)}q_1(T_0)\} \\
T_1^{(m)} = \arg\min_{T_1} \{L(T_1) - C^{(m-1)}q_0(T_1)\} \\
\text{Update cost as} \\
C^{(m)} = \frac{L(T_1^{(m)}) - L(T_0^{(m)})}{q_1(T_0^{(m)}) + q_0(T_1^{(m)})} \\
\text{until } C^{(m)} = C^{(m-1)} \]
AIFV-2 Construction Algorithm

- Yamamoto et al. proved that this Algorithm constructs optimal AIFV-2 Codes.
- At each step, algorithm creates two new improved code trees.
- Originally solved using ILP; later replaced by $O(n^5)$ DP algorithm. Parameterizes trees by “cost” $C$.

**Algorithm [Yamamoto et al]**

\[
m \leftarrow 0 \\
C^{(0)} = 2 - \log_2(3) \\
\text{repeat} \\
\quad m \leftarrow m + 1 \\
\quad T_0^{(m)} = \arg\min_{T_0} \{L(T_0) + C^{(m-1)}q_1(T_0)\} \\
\quad T_1^{(m)} = \arg\min_{T_1} \{L(T_1) - C^{(m-1)}q_0(T_1)\} \\
\text{Update cost as} \\
\quad C^{(m)} = \frac{L(T_1^{(m)}) - L(T_0^{(m)})}{q_1(T_0^{(m)}) + q_0(T_1^{(m)})} \\
\text{until } C^{(m)} = C^{(m-1)}
\]

They proved that Algorithm terminates after finite number of iterations, but no bound on number of iterations was known.
Outline

• Introduction

• AIFV-2 codes: cost and algorithm

• A Geometric Interpretation of the old algorithm
  • A New Binary Search Algorithm
  • An Ellipsoid Algorithm

• Extensions to AIFV-\( k \) codes (skip)

• Summing up and open questions
A Geometric Interpretation of the old algorithm

Algorithm [Yamamoto et al]

\[ m, C^{(0)} \leftarrow 0, 2 - \log_2(3) \]

repeat

\[ m \leftarrow m + 1 \]

\[ T^{(m)}_0 = \arg\min_{T_0} \{ L(T_0) + C^{(m-1)} q_1(T_0) \} \]

\[ T^{(m)}_1 = \arg\min_{T_1} \{ L(T_1) - C^{(m-1)} q_0(T_1) \} \]

\[ C^{(m)} = \frac{L(T^{(m)}_1) - L(T^{(m)}_0)}{q_1(T^{(m)}_0) + q_0(T^{(m)}_1)} \]

until \( C^{(m)} = C^{(m-1)} \)
A Geometric Interpretation of the old algorithm

Algorithm [Yamamoto et al]

$m, C^{(0)} \leftarrow 0, 2 - \log_2(3)$

repeat

\( m \leftarrow m + 1 \)

\( T_0^{(m)} = \arg\min_{T_0} \{ L(T_0) + C^{(m-1)} q_1(T_0) \} \)

\( T_1^{(m)} = \arg\min_{T_1} \{ L(T_1) - C^{(m-1)} q_0(T_1) \} \)

\( C^{(m)} = \frac{L(T_1^{(m)}) - L(T_0^{(m)})}{q_1(T_0^{(m)}) + q_0(T_1^{(m)})} \)

until \( C^{(m)} = C^{(m-1)} \)

• Original proof of termination was algebraic.
A Geometric Interpretation of the old algorithm

Algorithm [Yamamoto et al]

\[ m, C^{(0)} \leftarrow 0, 2 - \log_2(3) \]

repeat

\[ m \leftarrow m + 1 \]

\[ T^{(m)}_0 = \text{argmin}_{T_0} \{ L(T_0) + C^{(m-1)}q_1(T_0) \} \]

\[ T^{(m)}_1 = \text{argmin}_{T_1} \{ L(T_1) - C^{(m-1)}q_0(T_1) \} \]

\[ C^{(m)} = \frac{L(T^{(m)}_1) - L(T^{(m)}_0)}{q_1(T^{(m)}_0) + q_0(T^{(m)}_1)} \]

until \[ C^{(m)} = C^{(m-1)} \]

• Original proof of termination was algebraic.

• We replace algebraic viewpoint with a geometric one.
A Geometric Interpretation of the old algorithm

Algorithm [Yamamoto et al]

\[ m, C^{(0)} \leftarrow 0, 2 - \log_2(3) \]

repeat

\[ m \leftarrow m + 1 \]

\[ T_0^{(m)} = \arg\min_{T_0} \{ L(T_0) + C^{(m-1)}q_1(T_0) \} \]

\[ T_1^{(m)} = \arg\min_{T_1} \{ L(T_1) - C^{(m-1)}q_0(T_1) \} \]

\[ C^{(m)} = \frac{L(T_1^{(m)}) - L(T_0^{(m)})}{q_1(T_0^{(m)}) + q_0(T_1^{(m)})} \]

until \( C^{(m)} = C^{(m-1)} \)

• Original proof of termination was algebraic.

• We replace algebraic viewpoint with a geometric one.

For fixed \( T_0, T_1 \), these look like eqns of a line.

Eqn for \( x \)-coord of intersection of the 2 lines
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.

Construct the lower envelope $E_0$ of these lines. The optimization $\text{argmin}_{T_0} \{ L(T_0) + C^{(m-1)}q_1(T_0) \}$ in the algorithm finds the line $y_{T_0}(x)$ that corresponds to $E_0 \left( C^{(m-1)} \right)$.
Let $\mathcal{T}_0$ be the set of all possible code trees $T_0$. Then for all $T_0 \in \mathcal{T}_0$, the equation $y_{T_0}(x) = L(T_0) + xq_1(T_0)$ is a line with positive slope.

Construct the lower envelope $E_0$ of these lines. The optimization $\arg\min_{T_0} \{ L(T_0) + C^{(m-1)}q_1(T_0) \}$ in the algorithm finds the line $y_{T_0}(x)$ that corresponds to $E_0 \left( C^{(m-1)} \right)$. 
• Similarly, let $\mathcal{T}_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in \mathcal{T}_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.
Similarly, let \( T_1 \) be the set of all possible code trees \( T_1 \). Then for \( \forall T_1 \in T_1 \), the expression \( y_{T_1}(x) = L(T_1) - xq_0(T_1) \) is a line with negative slope.
Similarly, let $\mathcal{T}_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in \mathcal{T}_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.
Similarly, let $\mathcal{T}_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in \mathcal{T}_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.
Similarly, let $T_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in T_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.
Similarly, let $\mathcal{T}_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in \mathcal{T}_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.
Similarly, let $\mathcal{T}_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in \mathcal{T}_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.
Similarly, let $\mathcal{T}_1$ be the set of all possible code trees $T_1$. Then for $\forall T_1 \in \mathcal{T}_1$, the expression $y_{T_1}(x) = L(T_1) - xq_0(T_1)$ is a line with negative slope.

Construct the lower envelope $E_1$ of these lines. The optimization $\arg\min_{T_1} \{L(T_1) + C^{(m-1)}q_0(T_1)\}$ in the algorithm finds the $y_{T_1}(x)$ line that corresponds to $E_1(C^{(m-1)})$. 
- Because $E_0(x)$ has positive slope and $E_1(x)$ negative slope they intersect at a unique point $q$ with $x$-coordinate $x = C^*$. 
Geometric Interpretation of Algorithm

\[ C(i) \]

\[ y \]

\[ E_0(x) \]

\[ E_1(x) \]
Geometric Interpretation of Algorithm

At each step it uses DP algorithm to find the two lines $\ell_0(x)$ and $\ell_1(x)$ defining $E_0(x)$ and $E_1(x)$ at $x = C(i)$. 

![Diagram showing lines and functions](image-url)
Geometric Interpretation of Algorithm

At each step it uses DP algorithm to find the two lines $\ell_0(x)$ and $\ell_1(x)$ defining $E_0(x)$ and $E_1(x)$ at $x = C^{(i)}$.

It then finds the intersection point $p$ of $\ell_0(x)$ and $\ell_1(x)$ and sets $C^{(i+1)}$ to be the $x$-coordinate of that intersection point.
Geometric Interpretation of Algorithm

At each step it uses DP algorithm to find the two lines $\ell_0(x)$ and $\ell_1(x)$ defining $E_0(x)$ and $E_1(x)$ at $x = C^{(i)}$.

It then finds the intersection point $p$ of $\ell_0(x)$ and $\ell_1(x)$ and sets $C^{(i+1)}$ to be the $x$-coordinate of that intersection point.

Unless $p = q$, the unique intersection of $E_0(x)$ and $E_1(x)$, this process will continue, so it can only terminate if $C^{(i+1)} = C^*$.
A New Binary Search Algorithm

• This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.
A New Binary Search Algorithm

• This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.  

• Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^\ast$.
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.

Observation, $C^* \in [0, 1]$ and $C^* \in [l, r] \iff E_0(l) < E_1(l)$ and $E_1(r) < E_0(r)$
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.  
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.

Observation, $C^* \in [0, 1]$ and $C^* \in [l, r] \iff E_0(l) < E_1(l)$ and $E_1(r) < E_0(r)$
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.

Observation, $C^* \in [0, 1]$ and $C^* \in [l, r]$ $\iff$ $E_0(l) < E_1(l)$ and $E_1(r) < E_0(r)$
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.

Observation, $C^* \in [0, 1]$ and $C^* \in [l, r] \iff E_0(l) < E_1(l)$ and $E_1(r) < E_0(r)$
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^\ast$.
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.

Observation, $C^\ast \in [0, 1]$ and $C^\ast \in [l, r]$ $\iff$ $E_0(l) < E_1(l)$ and $E_1(r) < E_0(r)$
A New Binary Search Algorithm

- This geometric view permits replacing the iterative process with a simple binary search to find $C^*$.
- Works only for AIFV-2 (Not AIFV-$m$) but is very simple to understand.

- Observation, $C^* \in [0, 1]$ and $C^* \in [l, r]$ $\iff$ $E_0(l) < E_1(l)$ and $E_1(r) < E_0(r)$
• Theorem: If every probability $p_i$ is represented by at most $b$ bits, then if $r - l \leq 2^{-2b}$ the optimal solution $C^*$ can be found using with one more “query”.
• Theorem: If every probability $p_i$ is represented by at most $b$ bits, then if $r - l \leq 2^{-2b}$ the optimal solution $C^*$ can be found using with one more “query”.

• Proof in paper (standard techniques).
• Theorem: If every probability $p_i$ is represented by at most $b$ bits, then if $r - l \leq 2^{-2b}$ the optimal solution $C^*$ can be found using with one more “query”.
  
  • Proof in paper (standard techniques).

• After $O(\log(\frac{1}{2^{-2b}})) = O(b)$ queries, binary search can terminate.
• Theorem: If every probability $p_i$ is represented by at most $b$ bits, then if $r - l \leq 2^{-2b}$ the optimal solution $C^*$ can be found using with one more “query”.
  
  • Proof in paper (standard techniques).

• After $O(\log(\frac{1}{2^{-2b}})) = O(b)$ queries, binary search can terminate.

• In each query, the algorithm uses $O(n^5)$ time dynamic programming to find the trees (lines) on the lower envelopes for current value of $C$.

• Algorithm takes $O(n^5 b)$ time. This is first (weakly) polynomial algorithm for constructing AIFV-2 Codes.
An Ellipsoid Algorithm

- Although the binary search algorithm works for AIFV-2 codes, it does not generalize to AIFV-$m$ codes.
An Ellipsoid Algorithm

- Although the binary search algorithm works for AIFV-2 codes, it does not generalize to AIFV-$m$ codes.
- Need a stronger result from Convex Optimization due to Grotschel, Lovasz and Schrijver; the ellipsoid method.
An Ellipsoid Algorithm

- Although the binary search algorithm works for AIFV-2 codes, it does not generalize to AIFV-\(m\) codes.
- Need a stronger result from Convex Optimization due to Grotschel, Lovasz and Schrijver; the ellipsoid method.
- Let \(K\) be a convex set in \(\mathbb{R}^m\). A separation oracle for \(K\) is a procedure that, for any \(x \in \mathbb{R}^m\) either reports that \(x \in K\) or, if \(x \not\in K\), returns a hyperplane that separates \(x\) from \(K\).
An Ellipsoid Algorithm

- Although the binary search algorithm works for AIFV-2 codes, it does not generalize to AIFV-$m$ codes.
- Need a stronger result from Convex Optimization due to Grotschel, Lovasz and Schrijver; the ellipsoid method.
- Let $K$ be a convex set in $\mathbb{R}^m$. A separation oracle for $K$ is a procedure that, for any $x \in \mathbb{R}^m$ either reports that $x \in K$ or, if $x \notin K$, returns a hyperplane that separates $x$ from $K$.

- Ellipsoid Method: Let $K \in \mathbb{R}^m$ be a closed convex set and $c \in \mathbb{Q}^m$. Assume that we have a separation oracle for $K$. Also assume we know positive numbers $R$ and $\epsilon$ such that $K \subset B(0, R)$ and $Vol(K) > \epsilon$. Then with the ellipsoid method, in time polynomial in $m, \log \epsilon, \log R, \text{ and } \log \Delta$, we get a solution $x_0 \in K$ such that

$$c^T x_0 \geq \max\{c^T x | x \in K\} - \Delta |c|$$
The LP setup

- Where is the convex set $K$?
The LP setup

- Where is the convex set $K$?

$K$ is everything below both $E_0(x)$ and $E_1(x)$. Want to find $q$, highest point in $K$. 
• Where is the Separation Oracle?
- Where is the Separation Oracle?
- Known Dynamic Programming Algorithm! Returns the supporting lines of $E_0$ and $E_1$. Lower line either separates $p$ from $K$, or proves that $p \in K$. 

![Diagram showing $E_0(x)$ and $E_1(x)$ with a supporting line separating point $p$ from $K$.]
• Together the DP and the ellipsoid method lead to an $O(n^5b)$ time algorithm
• Together the DP and the ellipsoid method lead to an $O(n^5b)$ time algorithm

• For $m = 2$, run time no better than the binary search algorithm.
Together the DP and the ellipsoid method lead to an \( O(n^5b) \) time algorithm.

For \( m = 2 \), run time no better than the binary search algorithm.

However, algorithm works for constructing optimal AIFV-\( m \) codes (that use \( m \) coding trees).
Together the DP and the ellipsoid method lead to an \( O(n^5b) \) time algorithm.

For \( m = 2 \), run time no better than the binary search algorithm.

However, algorithm works for constructing optimal AIFV-\( m \) codes (that use \( m \) coding trees).

In \( m \)-ary case, AIFV-\( m \) codes construct \( m \) coding trees.
Encoding/decoding switches between trees.
Iterative algorithm for \( m = 2 \) case extends to general \( m \) case.
Similar to \( m = 2 \), it was unknown how many iterations were needed.

Binary searching technique cannot be applied but ellipsoid technique can. Leads to \( O(n^{2m+1}b) \) time algorithm.

Details in the paper.
Outline

• Introduction

• AIFV-2 codes: cost and algorithm

• A Geometric Interpretation of the old algorithm
  • A New Binary Search Algorithm
  • An Ellipsoid Algorithm

• Extensions to AIFV-$k$ codes (skip)

• Summing up and open questions
Summing up and open questions.

- Introduced idea of AIFV codes

- $O(n^5b)$ for AIFV-2 codes is still high. Can this be improved? Best known so far is $O(n^4b)$

- Are there strongly polynomial algorithms?

- Are there better AIFV codes? What is the tradeoff between number of coding trees used and compression? Everything known so far is empirical.