Dynamic Prefix Codes

Image Data Compression via Splay Trees and Plane-filling Curves

Mitchell Douglass Ryan Hermstein Colin Kincaid

Background: Huffman coding

- Well-known
- Weight-balanced frequency tree
- Statically optimal

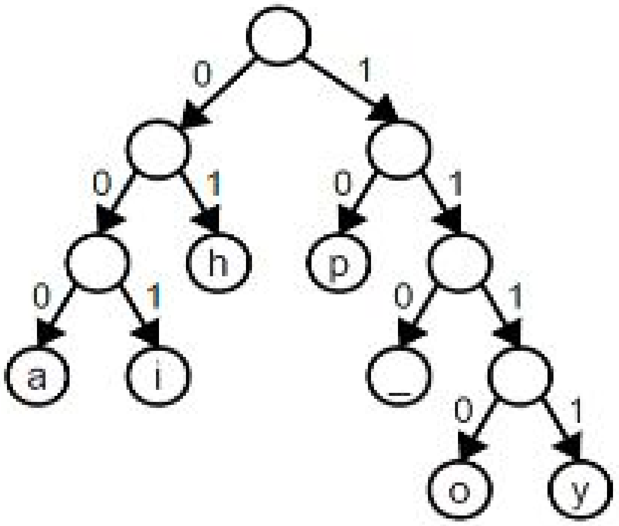


Diagram taken, with permission, from Dr. Cynthia Lee's CS106B, Spring 2016 lecture slides

Shannon entropy: What?

- A measure of redundancy in a file
  - How many bits are strictly needed per byte? (Ranges from 0 to 8)
- Calculated using the following formula:

$$H_p = \sum_{i=1}^{n} -p_i \lg p_i$$

(where p is the probability of a byte occurring)

- Note: We use the frequency of each byte as an estimate for p (self-entropy)

Shannon entropy: Oh, nice.

File	Shannon entropy	Huffman compression
google_home.txt	5.57	5.70
dictionary.txt	4.24	4.25
hamlet.txt	4.37	4.38

Shannon entropy: Yikes! Can we do better?

Fern: H = 7.447

Ryan: H = 7.468

City: H = 7.467

Dorm: H = 7.564

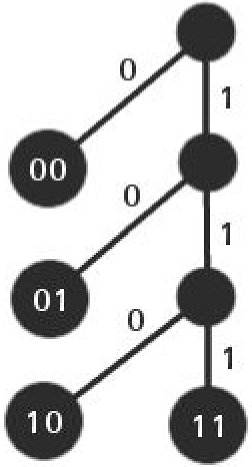
Building a Framework

Observation 1: A ***static prefix code*** (i.e. one that always encodes a specific message word by the same code word) corresponds to a unique binary tree with as many leaves as message words.

Building a Framework

Observation 1: A ***static prefix code*** (i.e. one that always encodes a specific message word by the same code word) corresponds to a unique binary tree with as many leaves as message words.

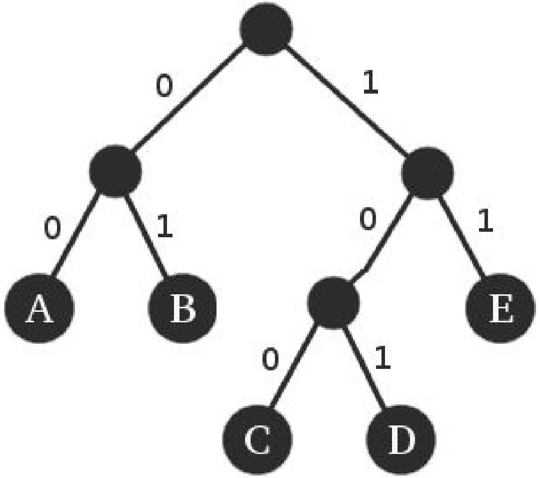
Msg. Word	Code Word
00	0
01	10
10	110
11	111



Building a Framework

Observation 1: A ***static prefix code*** (i.e. one that always encodes a specific message word by the same code word) corresponds to a unique binary tree with as many leaves as message words.

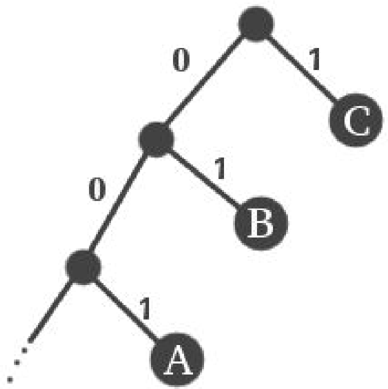
Msg. Word	**Code Word**
A	00
B	01
C	100
D	101
E	11



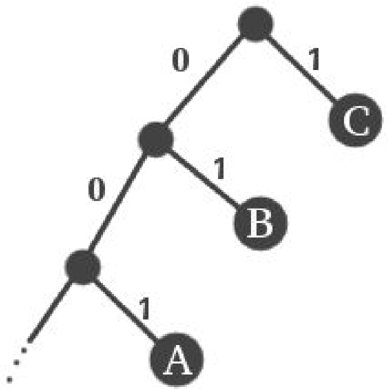
Building a Framework

Observation 2: The encoding/decoding algorithms of Huffman do not break down if we use a dynamic binary tree in place of a weight-balanced binary tree. Since root-leaf paths are communicated in the code words, encoding/decoding can remain in sync.

Encoder



Decoder

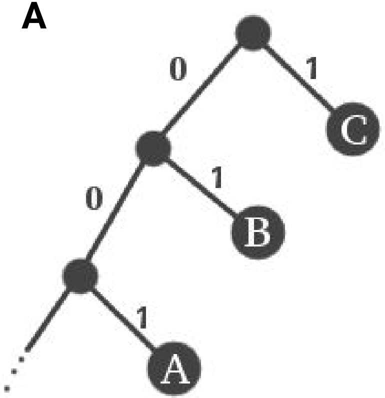


Building a Framework

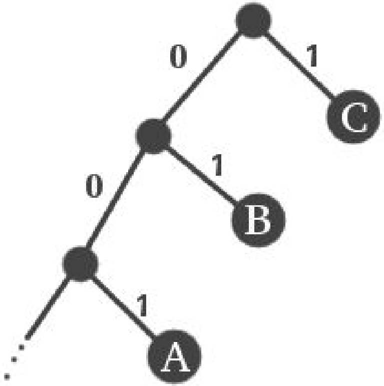
Observation 2: The encoding/decoding algorithms of Huffman do not break down if we use a dynamic binary tree in place of a weight-balanced binary tree. Since root-leaf paths are communicated in the code words, encoding/decoding can remain in sync.

A

Encoder

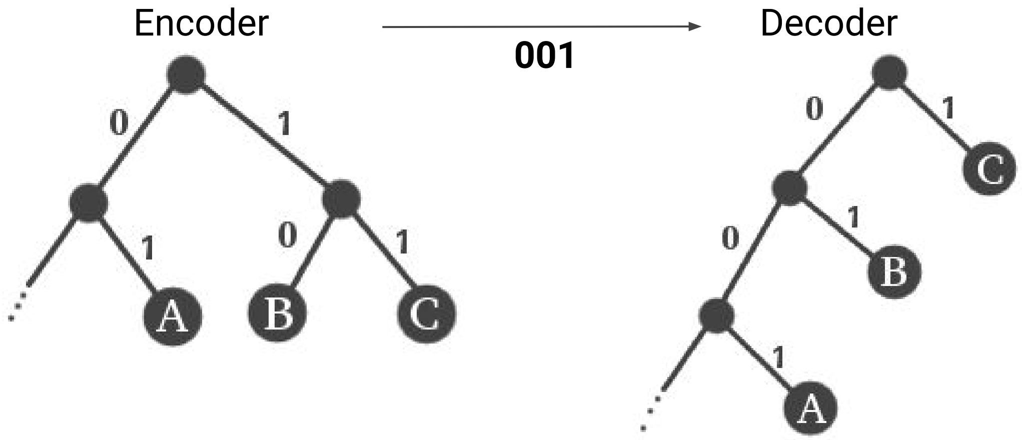


Decoder



Building a Framework

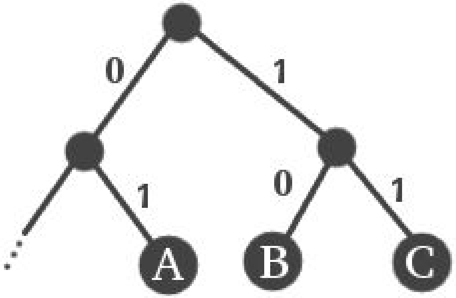
Observation 2: The encoding/decoding algorithms of Huffman do not break down if we use a dynamic binary tree in place of a weight-balanced binary tree. Since root-leaf paths are communicated in the code words, encoding/decoding can remain in sync.



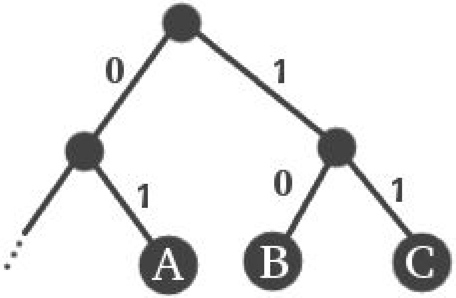
Building a Framework

Observation 2: The encoding/decoding algorithms of Huffman do not break down if we use a dynamic binary tree in place of a weight-balanced binary tree. Since root-leaf paths are communicated in the code words, encoding/decoding can remain in sync.

Encoder



Decoder



The Framework Dynamic Prefix Codes

Framework: Suppose we have access to the following:

1. A **Digest Function**, which produces message digests of size proportional to the message alphabet.
2. An **Initialization Function**, that produces an initial binary tree, given a message digest.
3. A **Dynamic Tree Strategy**, for performing restructuring to a binary tree, given a path from the root to a leaf node of the tree.

Then we can build a dynamic prefix code using an encoder/decoder in the style of the Huffman code.

Splay Trees

- Splay Trees are a dynamic binary tree strategy in which nodes are rotated along the root-node path of any node access. We studied these restructuring rules in lecture.
- Can we use Splay Trees with our framework for dynamic prefix codes?

- **Problem:** Splay Trees move accessed nodes up to the root of the tree, but our framework requires binary trees with message words stored in the leaves. Can we fix this?

Splay Modifications

- Key differences when working with prefix codes
  - Accessed nodes need to remain leaves
  - Moving internal nodes is unnecessary because they store no data
- Solution: Semisplaying
  - Target node's children are not modified
  - Path from the root to the target is shortened by a factor of two

(b)

Splay Modifications

- Semisplaying achieves the same theoretical bounds as splaying within a constant factor
- There are other optimizations to simplify the splaying, but this is the main change
- See our paper for more details!

Splay Tree Properties

Theorem (Balance Theorem): A sequence of ***m*** element accesses on a splay tree of **n** keys requires total amortized time **O(m log n + n log n)**.

Implication: a dynamic prefix code based on the splay tree will produce message encodings that are within a constant factor of an encoding **based on a fixed length code**.

There is no pathological data sequence producing arbitrarily large encodings.

Splay Tree Properties

Theorem (Static Optimality Theorem): A sequence of ***m*** element accesses on a splay tree of **n** keys, where each key is accessed at least once, requires an average amortized time of **O(1 + H)** per access, where **H** is the Shannon Entropy of the sequence.

Implication: a dynamic prefix code based on the splay tree will produce message encodings that are within a constant factor of an encoding **based on the Huffman Code**.

The Huffman code does not beat a splay prefix code by more than a constant factor.

Splay Tree Properties

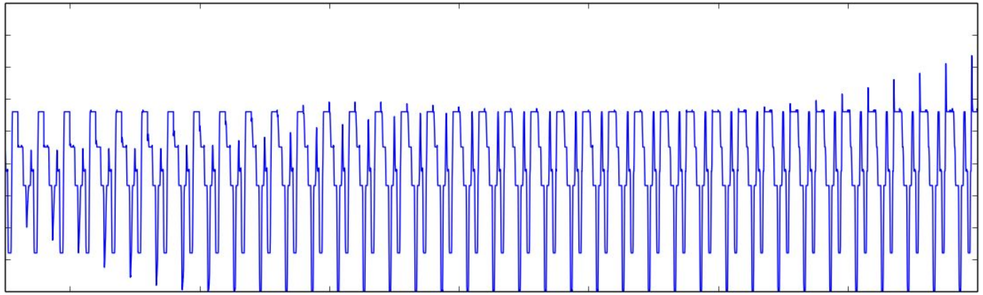
Theorem (Working Set Theorem): A sequence of ***m*** element accesses on a splay tree of **n** keys, where each key is accessed at least once, requires an amortized time of **O(1 + log t(x))** on access of **x**, where **t(x)** is the number of unique elements accessed since the last access of **x**.

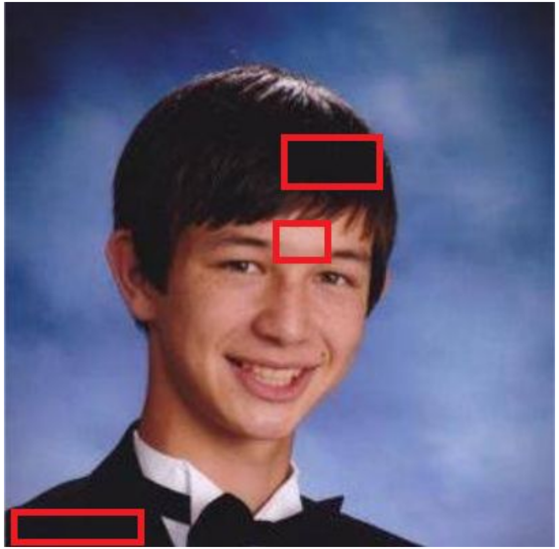
Implication: a dynamic prefix code based on the splay tree can take advantage of temporal state in a data stream.

E.g. An extended subsequence consisting of only 16 unique byte values will be encoded by ~4 bits in the worst case.

Images, Revisited

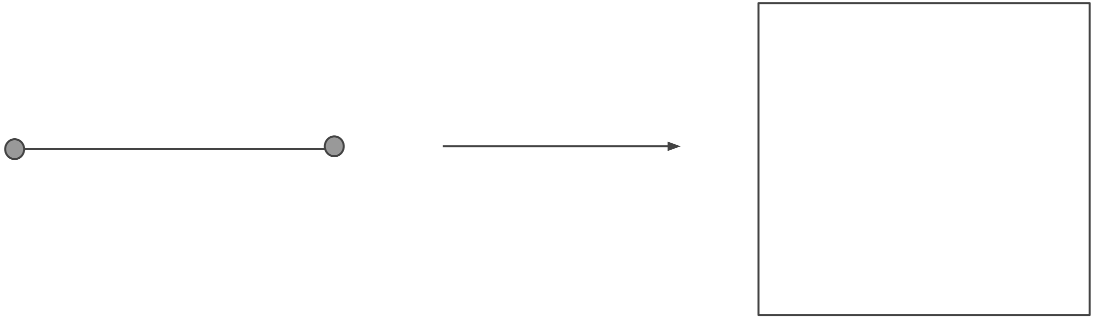
- It is not hard to see that images exhibit a sort of "Working Set" property, just look at these isolated regions!
- Standard Row-Order Enumerations of Images do not exploit the locality of these regions well.





Plane-Filling Curves

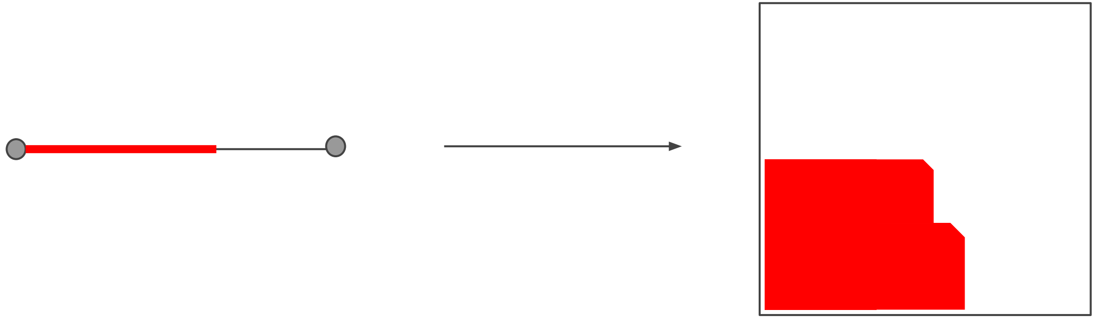
Definition: A **plane-filling curve** is a continuous map from the unit interval [0, 1] to the unit square [0, 1]x[0, 1]



For our purposes, plane-filling curves help us define an enumeration of pixels within an image, such that consecutive pixels are neighbors in the image.

Plane-Filling Curves

Definition: A **plane-filling curve** is a continuous map from the unit interval [0, 1] to the unit square [0, 1]x[0, 1]

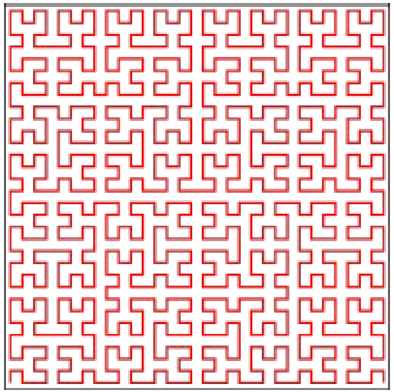


For our purposes, plane-filling curves help us define an enumeration of pixels within an image, such that consecutive pixels are neighbors in the image.

The Hilbert Curve

2	3
1	4

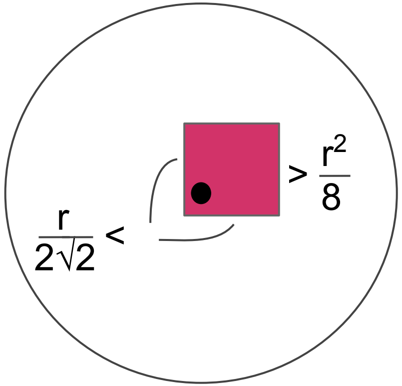
6	7	10	11
5	8	9	12
4	3	14	13
1	2	15	16



The Hilbert Curve

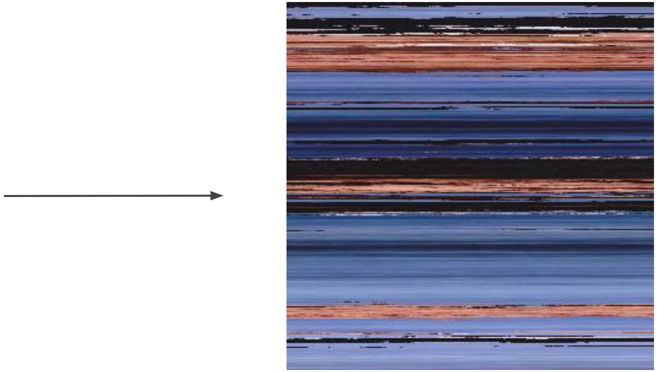
- The Hilbert curve is especially good for our purposes. It has the following property

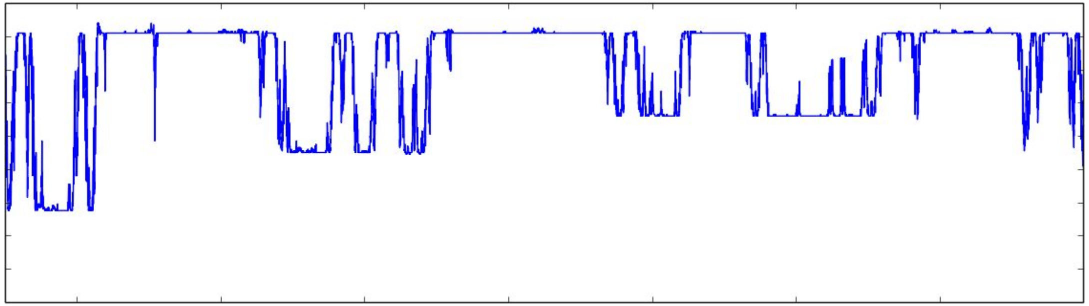
Proposition: Given an enumeration of the pixels within an image given by the Hilbert Curve, the center pixel of a circular region of pixels of radius **r** is contained in a sequence of pixels of length at least $\mathbf{r^2/3}$.



Compare this with the guarantee of **2r** with row-order traversal. For **r > 5**, Hilbert curve enumerations produce longer "working set" sequences.

Some Empirical Evidence





Results

- This experiment compared bits/byte ratios on a variety of images
- Splay is able to outperform Huffman on most images
- There are sufficient regions of repeated pixels for Splay to gain this advantage
- Huffman outperforms Splay on the randomly generated images

Image File	Method	size (kB)	$\left(\frac{\text{bits}}{\text{byte}}\right)$
random	Huff	750	(8.00)
(750 KB)	MTF	1344	(14.3)
H = 8.000	Splay	906	(9.66)
expRandom	Huff	187	(1.99)
(750 KB)	MTF	261	(2.78)
H = 1.989	Splay	216	(2.30)
ryan	Huff	253	(7.50)
(270 KB)	MTF	265	(7.85)
H = 7.468	Splay	202	(5.99)
fern	Huff	4779	(7.47)
(5117 KB)	MTF	5574	(8.71)
H = 7.447	Splay	4267	(6.67)
city	Huff	5614	(7.50)
(5989 KB)	MTF	5176	(6.91)
H = 7.467	Splay	3806	(5.08)
dorm	Huff	500	(7.59)
(527 KB)	MTF	480	(7.29)
H = 7.564	Splay	372	(5.65)

Results

- The next experiment used images **with PFC**
- Splay is adaptive and benefits greatly from this change
  - Decrease by about 2 in bits/byte ratio!
- Now Splay significantly outperforms Huffman in the non-random images

Image File	Method	size (kB)	$(\frac{\text{bits}}{\text{byte}})$
ryan	Huff	253	(7.50)
(270 KB)	MTF	188	(5.57)
H = 7.468	Splay	152	(4.50)
fern	Huff	4779	(7.47)
(5117 KB)	MTF	4431	(6.93)
H = 7.447	Splay	3347	(5.23)
city	Huff	5614	(7.50)
(5989 KB)	MTF	4682	(6.25)
H = 7.467	Splay	3427	(4.58)
dorm	Huff	500	(7.59)
(527 KB)	MTF	320	(4.86)
H = 7.564	Splay	269	(4.08)

Future Work

- Lossy compression
- Alternative pixel representations

Questions?