15-853: Algorithms in the Real World

Data Compression 4
Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, ...
Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
Applications of Probability Coding: PPM + others
Lempel-Ziv Algorithms: LZ77, gzip, compress, ...
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
  - Scalar and vector quantization
  - JPEG and MPEG
Compressing graphs and meshes: BBK
Scalar Quantization

Quantize regions of values into a single value:

Can be used to reduce # of bits for a pixel
Vector Quantization

1. **Generate Vector**
   - In
   - **Generate Output**
   - Out

2. **Encode**
   - Find closest code vector
   - **Index**
   - **Codebook**

3. **Decode**
   - **Index**
   - **Codebook**

Generate Output

Vector Quantization
Vector Quantization

What do we use as vectors?
• Color (Red, Green, Blue)
  - Can be used, for example to reduce 24bits/pixel to 8bits/pixel
  - Used in some terminals to reduce data rate from the CPU (colormaps)
• K consecutive samples in audio
• Block of K pixels in an image

How do we decide on a codebook
• Typically done with clustering
Vector Quantization: Example
Linear Transform Coding

Want to encode values over a region of time or space
- Typically used for images or audio

Select a set of linear basis functions $\phi_i$ that span the space
- sin, cos, spherical harmonics, wavelets, ...
- Defined at discrete points
Linear Transform Coding

Coefficients: \( \Theta_i = \sum_j x_j \phi_i(j) = \sum_j x_j a_{ij} \)

\( \Theta_i = i^{th} \) resulting coefficient

\( x_j = j^{th} \) input value

\( a_{ij} = ij^{th} \) transform coefficient = \( \phi_i(j) \)

In matrix notation:

\[ \Theta = Ax \]

\[ x = A^{-1} \Theta \]

Where \( A \) is an \( n \times n \) matrix, and each row defines a basis function
Example: Cosine Transform

\[ \Theta_i = \sum_j x_j \phi_i(j) \]
Other Transforms

Polynomial:

- $1$
- $x$
- $x^2$

Wavelet (Haar):
How to Pick a Transform

Goals:

- Decorrelate
- Low coefficients for many terms
- Basis functions that can be ignored by perception

Why is using a Cosine of Fourier transform across a whole image bad?

How might we fix this?
Usefulness of Transform

Typically transforms $A$ are orthonormal: $A^{-1} = A^T$

Properties of orthonormal transforms:
- $\sum x^2 = \sum \Theta^2$ (energy conservation)

Would like to compact energy into as few coefficients as possible

$$G_{TC} = \frac{1}{n} \sum \sigma_i^2 \left( \prod \sigma_i^2 \right)^{-\frac{1}{n}}$$

(the **transform coding gain**)

arithmetic mean/geometric mean

$$\sigma_i = (\Theta_i - \Theta_{av})$$

The higher the gain, the better the compression
Case Study: JPEG

A nice example since it uses many techniques:
- Transform coding (Cosine transform)
- Scalar quantization
- Difference coding
- Run-length coding
- Huffman or arithmetic coding

JPEG (Joint Photographic Experts Group) was designed in 1991 for lossy and lossless compression of color or grayscale images. The lossless version is rarely used.

Can be adjusted for compression ratio (typically 10:1)
JPEG in a Nutshell

![Diagram of JPEG compression process]

- RGB images are converted to YIQ format (optional).
- Each plane is encoded separately.
- Quantization and Discrete Cosine Transform (DCT) are applied to each 8x8 block.
- Zig-zag ordering of the DCT coefficients is performed.
- DC difference from the previous block is encoded.
- Run-length encoding (RLE) or Huffman/Arithmetic coding is used to compress the data.
- The compressed data is then transmitted or stored.
JPEG: Quantization Table

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>11</th>
<th>10</th>
<th>16</th>
<th>24</th>
<th>40</th>
<th>51</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<td>19</td>
<td>26</td>
<td>58</td>
<td>60</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>14</td>
<td>16</td>
<td>24</td>
<td>40</td>
<td>57</td>
<td>69</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>22</td>
<td>29</td>
<td>51</td>
<td>87</td>
<td>80</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>22</td>
<td>37</td>
<td>56</td>
<td>68</td>
<td>109</td>
<td>103</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>35</td>
<td>55</td>
<td>64</td>
<td>81</td>
<td>104</td>
<td>113</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>64</td>
<td>78</td>
<td>87</td>
<td>103</td>
<td>121</td>
<td>120</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>92</td>
<td>95</td>
<td>98</td>
<td>112</td>
<td>100</td>
<td>103</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

Also divided through uniformly by a quality factor which is under control.
JPEG: Block scanning order

Uses run-length coding for sequences of zeros
JPEG: example

.125 bits/pixel (factor of 200)
Case Study: MPEG

Pretty much JPEG with **interframe coding**

Three types of frames

- **I** = intra frame (aprox. JPEG) anchors
- **P** = predictive coded frames
- **B** = bidirectionally predictive coded frames

**Example:**

<table>
<thead>
<tr>
<th>Type</th>
<th>I</th>
<th>B</th>
<th>B</th>
<th>P</th>
<th>B</th>
<th>B</th>
<th>P</th>
<th>B</th>
<th>B</th>
<th>P</th>
<th>B</th>
<th>B</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>

**I** frames are used for random access.
MPEG matching between frames

Diagram showing MPEG matching process between a target and reference frame, with steps involving difference, best match motion vector, DCT + Quan + RLE, Huffman coder, and resulting code 0100110.
MPEG: Compression Ratio

356 x 240 image

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>18KB</td>
<td>7/1</td>
</tr>
<tr>
<td>P</td>
<td>6KB</td>
<td>20/1</td>
</tr>
<tr>
<td>B</td>
<td>2.5KB</td>
<td>50/1</td>
</tr>
<tr>
<td>Average</td>
<td>4.8KB</td>
<td>27/1</td>
</tr>
</tbody>
</table>

30 frames/sec x 4.8KB/frame x 8 bits/byte
= 1.2 Mbits/sec + .25 Mbits/sec (stereo audio)

HDTV has 15x more pixels
= 18 Mbits/sec
MPEG in the “real world”

• DVDs
  - Adds “encryption” and error correcting codes
• Direct broadcast satellite
• HDTV standard
  - Adds error correcting code on top
• Storage Tech “Media Vault”
  - Stores 25,000 movies

Encoding is much more expensive than encoding. Still requires special purpose hardware for high resolution and good compression.
Wavelet Compression

- A set of localized basis functions
- Avoids the need to block

“mother function” $\varphi(x)$

$\varphi_{sl}(x) = \varphi(2^s x - l)$

$s =$ scale $\quad l =$ location

Requirements

\[
\int_{-\infty}^{\infty} \varphi(x) dx = 0 \quad \text{and} \quad \int_{-\infty}^{\infty} |\varphi(x)|^2 dx < \infty
\]

Many mother functions have been suggested.
Haar Wavelets

Most described, least used.

\[ \varphi(x) = \begin{cases} 
1 & 0 \leq x < 1/2 \\
-1 & 1/2 \leq x < 1 \\
0 & \text{otherwise}
\end{cases} \]

\[ H_{s\ell}(x) = \varphi(2^s x - \ell) \]

+ DC component = \( 2^{k+1} \) components
Haar Wavelet in 2d
Discrete Haar Wavelet Transform

How do we convert this to the wavelet coefficients?

\[
\begin{align*}
H_{00} & : 0 & 1 \\
H_{10} & : 0.5 & 1 \\
H_{20} & : 0.25 & 0.5 & 0.75 & 1 \\
\end{align*}
\]
Discrete Haar Wavelet Transform

for (j = n/2; j >= 1; j = j/2) {
    for (i = 1; i < j; i++) {
        b[i] = (a[2i-1] + a[2i])/2;
        b[j+i] = (a[2i-1] - a[2i])/2;
    }
    a[1..2*j] = b[1..2*j];
}

How do we convert this to the wavelet coefficients?

Averages

Differences

Linear time!
Haar Wavelet Transform: example

\[ a = \begin{array}{cccccccccc}
2 & 1 & 2 & -1 & -2 & 0 & 2 & -2 \\
1.5 & .5 & -1 & 0 & .5 & 1.5 & -1 & 2 \\
1 & -.5 & .5 & -.5 \\
.25 & .75 \\
\end{array} \]

\[ a = \begin{array}{cccccccccc}
.25 & .75 & .5 & .5 & .5 & 1.5 & -1 & 2 \\
\end{array} \]
Wavelet decomposition
Morlet Wavelet

\[ \phi(x) = \text{Gaussian} \times \cosine = e^{-\left(\frac{x^2}{2}\right)} \cos(5x) \]

Corresponds to wavepackets in physics.
Daubechies Wavelet
JPEG2000

Overall Goals:

- High compression efficiency with good quality at compression ratios of .25bpp
- Handle large images (up to $2^{32} \times 2^{32}$)
- Progressive image transmission
  - Quality, resolution or region of interest
- Fast access to various points in compressed stream
- Pan and Zoom while only decompressing parts
- Error resilience
JPEG2000: Outline

Main similarities with JPEG
• Separates into Y, I, Q color planes, and can downsample the I and Q planes
• Transform coding

Main differences with JPEG
• Wavelet transform
  - Daubechies 9-tap/7-tap (irreversible)
  - Daubechies 5-tap/3-tap (reversible)
• Many levels of hierarchy (resolution and spatial)
• Only arithmetic coding
JPEG2000: 5-tap/3-tap

\[ h[i] = a[2i-1] - \frac{(a[2i] + a[2i-2])}{2}; \]
\[ l[i] = a[2i] + \frac{(h[i-1] + h[i] + 2)}{2}; \]

\( h[i] \): is the “high pass” filter, ie, the **differences**
  it depends on 3 values from a (3-tap)
\( l[i] \): is the “low pass” filter, ie, the **averages**
  it depends on 5 values from a (5-tap)

Need to deal with boundary effects.
This is reversible: assignment
JPEG 2000: Outline

A spatial and resolution hierarchy

- **Tiles:** Makes it easy to decode sections of an image. For our purposes we can imagine the whole image as one tile.

- **Resolution Levels:** These are based on the wavelet transform. High-detail vs. Low detail.

- **Precinct Partitions:** Used within each resolution level to represent a region of space.

- **Code Blocks:** blocks within a precinct

- **Bit Planes:** ordering of significance of the bits
JPEG2000: Precincts
JPEG vs. JPEG2000

JPEG: .125bpp

JPEG2000: .125bpp
Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, ...
Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
Applications of Probability Coding: PPM + others
Lempel-Ziv Algorithms: LZ77, gzip, compress, ...
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK
Compressing Structured Data

So far we have concentrated on Text and Images, compressing sound is also well understood. What about various forms of “structured” data?

- Web indexes
- Triangulated meshes used in graphics
- Maps (mapquest on a palm)
- XML
- Databases
Compressing Graphs

**Goal:** To represent large graphs compactly while supporting queries efficiently
- e.g., adjacency and neighbor queries
- want to do significantly better than adjacency lists (e.g. a factor of 10 less space, about the same time)

**Applications:**
- Large web graphs
- Large meshes
- Phone call graphs
How to start?

**Lower bound** for $n$ vertices and $m$ edges?

1. If there are $N$ possible graphs then we will need $\log N$ bits to distinguish them.

2. In a directed graph there are $n^2$ possible edges (allowing self edges).

3. We can choose any $m$ of them so $N = \binom{n^2}{m}$.

4. We will need $\log (\binom{n^2}{m}) = O(m \log (n^2/m))$ bits in general.

For sparse graphs ($m = kn$) this is hardly any better than adjacency lists (perhaps factor of 2 or 3).
What now?

Are all graphs equally likely?
Are there properties that are common across “real world” graphs?

Consider
- link graphs of the web pages
- map graphs
- router graphs of the internet
- meshes used in simulations
- circuit graphs

LOCAL CONNECTIONS / SMALL SEPARATORS
Edge Separators

An edge separator for \((V,E)\) is a set of edges \(E' \subseteq E\) whose removal partitions \(V\) into two components \(V_1\) and \(V_2\).

Goals:
- **balanced** \(|V_1| = \frac{1}{4} |V_2|\)
- **small** \(|E'|\) is small

A class of graphs \(S\) satisfies a \(f(n)\)-edge separator theorem if

9 \(\alpha < 1, \beta > 0\)
8 \((V,E) \in S, 9\) separator \(E'\),
\(|E'| < \beta f(|V|),\)
\(|V_i| < \alpha |V|, i = 1,2\)

Can also define vertex separators.
Separable Classes of Graphs

Planar graphs: $O(n^{1/2})$ separators
Well-shaped meshes in $\mathbb{R}^d$: $O(n^{1-1/d})$ [Miller et al.]
Nearest-neighbor graphs
In practice, good separators from circuit graphs, street graphs, web connectivity graphs, router connectivity graphs

Note: All separable classes of graphs have bounded density ($m$ is $O(n)$)
Main Ideas

- Number vertices so adjacent vertices have similar numbers
  - Use separators to do this
- Use difference coding on adjacency lists
- Use efficient data structure for indexing
Compressed Adjacency Tables

```
<table>
<thead>
<tr>
<th>#</th>
<th>D</th>
<th>Neighbors</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3 4</td>
<td>3 1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4 5 6 8</td>
<td>3 1 1 2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0 7</td>
<td>-3 7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0 1 7 8</td>
<td>-4 1 6 1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1 6 8</td>
<td>-4 5 2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1 2 5</td>
<td>-5 1 3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3 4 8</td>
<td>-4 1 4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1 4 5 7</td>
<td>-7 3 1 2</td>
</tr>
</tbody>
</table>
```
Compressed Adjacency Tables

![Graph Diagram]

<table>
<thead>
<tr>
<th>#</th>
<th>D</th>
<th>Neighbors</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0 2 3</td>
<td>-1 2 1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1 3 4</td>
<td>-1 2 1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1 2 4 5</td>
<td>-1 1 2 1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2 3 5 6</td>
<td>-2 1 2 1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3 4 6 7</td>
<td>-2 1 2 1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4 5 8</td>
<td>-2 1 3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5 8</td>
<td>-2 3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>6 7</td>
<td>-2 1</td>
</tr>
</tbody>
</table>
Log-sized Codes

Log-sized code: Any prefix code that takes $O(\log (d))$ bits to represent an integer $d$.

Gamma code, delta code, skewed Bernoulli code

Example: Gamma code

Prefix: unary code for $\lfloor \log d \rfloor$

Suffix: binary code for $d-2^{\lfloor \log d \rfloor}$

(binary code for $d$, except leading 1 is implied)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>01 0</td>
</tr>
<tr>
<td>3</td>
<td>01 1</td>
</tr>
<tr>
<td>4</td>
<td>001 00</td>
</tr>
<tr>
<td>5</td>
<td>001 01</td>
</tr>
<tr>
<td>6</td>
<td>001 10</td>
</tr>
<tr>
<td>7</td>
<td>001 11</td>
</tr>
<tr>
<td>8</td>
<td>0001 000</td>
</tr>
</tbody>
</table>
Difference Coding

For each vertex, encode:
- Degree
- Sign of first entry
- Differences in adjacency list

Concatenate vertex encodings to encode the graph

<table>
<thead>
<tr>
<th>#</th>
<th>D</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010 0 011 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>degree sign 3 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>D</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>-4 1 6 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00100 1 00100 1 00110 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>degree sign 4 1 6 1</td>
</tr>
</tbody>
</table>

Renumbering with Edge Separators
Renumbering with Edge Separators
Renumbering with Edge Separators
Renumbering with Edge Separators
Theorem (edge separators)

Any class of graphs that allows $O(n^c)$ edge separators can be compressed to $O(n)$ bits with $O(1)$ access time using:
- Difference coded adjacency lists
- $O(n)$-bit indexing structure
**Performance: Adjacency Table**

<table>
<thead>
<tr>
<th></th>
<th>dfs</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
</tr>
<tr>
<td>auto</td>
<td>0.79</td>
<td>9.88</td>
<td>153.11</td>
<td>5.17</td>
<td>7.54</td>
<td>5.90</td>
<td>14.59</td>
</tr>
<tr>
<td>feocean</td>
<td>0.06</td>
<td>13.88</td>
<td>388.83</td>
<td>7.66</td>
<td>17.16</td>
<td>8.45</td>
<td>34.83</td>
</tr>
<tr>
<td>m14b</td>
<td>0.31</td>
<td>10.65</td>
<td>181.41</td>
<td>4.81</td>
<td>8.16</td>
<td>5.45</td>
<td>15.32</td>
</tr>
<tr>
<td>ibm17</td>
<td>0.44</td>
<td>13.01</td>
<td>136.43</td>
<td>6.18</td>
<td>11.0</td>
<td>6.79</td>
<td>20.25</td>
</tr>
<tr>
<td>ibm18</td>
<td>0.48</td>
<td>11.88</td>
<td>129.22</td>
<td>5.72</td>
<td>9.5</td>
<td>6.24</td>
<td>17.29</td>
</tr>
<tr>
<td>CA</td>
<td>0.76</td>
<td>8.41</td>
<td>382.67</td>
<td>4.38</td>
<td>14.61</td>
<td>4.90</td>
<td>35.21</td>
</tr>
<tr>
<td>PA</td>
<td>0.43</td>
<td>8.47</td>
<td>364.06</td>
<td>4.45</td>
<td>13.95</td>
<td>4.98</td>
<td>33.02</td>
</tr>
<tr>
<td>googleI</td>
<td>1.4</td>
<td>7.44</td>
<td>186.91</td>
<td>4.08</td>
<td>12.71</td>
<td>4.18</td>
<td>40.96</td>
</tr>
<tr>
<td>googleO</td>
<td>1.4</td>
<td>11.03</td>
<td>186.91</td>
<td>6.78</td>
<td>12.71</td>
<td>6.21</td>
<td>40.96</td>
</tr>
<tr>
<td>lucent</td>
<td>0.04</td>
<td>7.56</td>
<td>390.75</td>
<td>5.52</td>
<td>19.5</td>
<td>5.54</td>
<td>45.75</td>
</tr>
<tr>
<td>scan</td>
<td>0.12</td>
<td>8.00</td>
<td>280.25</td>
<td>5.94</td>
<td>23.33</td>
<td>5.76</td>
<td>81.75</td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td><strong>10.02</strong></td>
<td><strong>252.78</strong></td>
<td><strong>5.52</strong></td>
<td><strong>13.65</strong></td>
<td><strong>5.86</strong></td>
<td><strong>34.54</strong></td>
<td><strong>5.56</strong></td>
</tr>
</tbody>
</table>

Time is to create the structure, normalized to time for DFS
# Performance: Overall

<table>
<thead>
<tr>
<th>Graph</th>
<th>Array</th>
<th>List</th>
<th>bu-cf/semi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>space</td>
<td>time</td>
</tr>
<tr>
<td>auto</td>
<td>0.24</td>
<td>34.2</td>
<td>0.61</td>
</tr>
<tr>
<td>feocean</td>
<td>0.04</td>
<td>37.6</td>
<td>0.08</td>
</tr>
<tr>
<td>m14b</td>
<td>0.11</td>
<td>34.1</td>
<td>0.29</td>
</tr>
<tr>
<td>ibm17</td>
<td>0.15</td>
<td>33.3</td>
<td>0.40</td>
</tr>
<tr>
<td>ibm18</td>
<td>0.14</td>
<td>33.5</td>
<td>0.38</td>
</tr>
<tr>
<td>CA</td>
<td>0.34</td>
<td>43.4</td>
<td>0.56</td>
</tr>
<tr>
<td>PA</td>
<td>0.19</td>
<td>43.3</td>
<td>0.31</td>
</tr>
<tr>
<td>googleI</td>
<td>0.24</td>
<td>37.7</td>
<td>0.49</td>
</tr>
<tr>
<td>googleO</td>
<td>0.24</td>
<td>37.7</td>
<td>0.50</td>
</tr>
<tr>
<td>lucent</td>
<td>0.02</td>
<td>42.0</td>
<td>0.04</td>
</tr>
<tr>
<td>scan</td>
<td>0.04</td>
<td>43.4</td>
<td>0.06</td>
</tr>
</tbody>
</table>

*Time is for one DFS*
Conclusions

O(n)-bit representation of separable graphs with $O(1)$-time queries
Space efficient and fast in practice for a wide variety of graphs.
Compression Summary

Compression is all about **probabilities**

We want the model to skew the probabilities as much as possible (i.e., decrease the **entropy**).
Compression Summary

How do we figure out the probabilities
- Transformations that skew them
  - Guess value and code difference
  - Move to front for temporal locality
  - Run-length
  - Linear transforms (Cosine, Wavelet)
  - Renumber (graph compression)
- Conditional probabilities
  - Neighboring context

In practice one almost always uses a combination of techniques