Peeling Arguments
Invertible Bloom Lookup Tables
and Biff Codes

Michael Mitzenmacher
• Survey some of the history of peeling arguments.
A Matching Peeling Argument

• Take a random graph with $n$ vertices and $m$ edges.

• Form a matching greedily as follows:
  – Find a vertex of degree 1.
  – Put the corresponding edge in the matching.
  – Repeat until out of vertices of degree 1.
History

• Studied by Karp and Sipser in 1981.
• Threshold behavior: showed that if $m < en/2$, then the number of leftover edges is $o(n)$.
• Used an analysis based on differential equations.
A SAT Peeling Argument

• Random kSAT and the *pure literal rule*

\[(x_1 \lor x_2 \lor \overline{x}_4) \land (\overline{x}_2 \lor \overline{x}_3 \lor \overline{x}_4) \land (\overline{x}_1 \lor \overline{x}_3 \lor \overline{x}_4) \ldots\]

  – Typical setting: m clauses, n variables, k literals per clause

• Greedy algorithm:
  – While there exists a literal that appears 0 times, set its value to 0 (and its negation to 1).
  – Peel away any clauses with its negation.
  – Continue until done.
Random Graph Interpretation

- Bipartite graph with vertices for literal pairs on one side, vertices for clauses on the other.
- When neighbors for one literal in a pair drops to 0, can remove both of them, and neighboring clauses.
History

- Put in fluid limit/differential equations framework by Mitzenmacher.
  - Threshold: when $m/n < c_k$, solution found with high probability.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.636938...</td>
</tr>
<tr>
<td>4</td>
<td>1.544559...</td>
</tr>
<tr>
<td>5</td>
<td>1.403560...</td>
</tr>
<tr>
<td>6</td>
<td>1.274162...</td>
</tr>
<tr>
<td>7</td>
<td>1.163550...</td>
</tr>
<tr>
<td>8</td>
<td>1.069994...</td>
</tr>
<tr>
<td>9</td>
<td>0.990510...</td>
</tr>
<tr>
<td>10</td>
<td>0.922394...</td>
</tr>
</tbody>
</table>
A Peeling Paradigm

• Start with a random graph
• Peel away part of the graph greedily.
  – Generally, find a node of degree 1, remove the edge and the other adjacent node, but there are variations.
    • k-core = maximal subgraph with degree at least k.
    • Find vertex with degree less than k, remove it and all edges, continue.

• Analysis approach:
  – Left with a random graph (with a different degree distribution).
  – Keep reducing the graph all the way down.
    • To an empty graph, or a stopping point.
Not Just for Theory

• Applications to codes, hash-based data structures.
  – These ideas keep popping up.
• Resulting algorithms and data structures are usually very efficient.
  – Simple greedy approach.
  – Low overhead accounting (counts on vertices).
  – Often yields linear or quasi-linear time.
• Useful for *big data* applications.
Low Density Parity Check Codes
Decoding by Peeling

\[ a \longrightarrow \quad b \]
\[ ? \longrightarrow \quad b \oplus g \]
\[ c \longrightarrow \quad e \oplus g \oplus h \]
\[ d \longrightarrow \quad b \oplus e \oplus g \oplus h \]
\[ ? \longrightarrow \quad f \]
\[ ? \longrightarrow \quad ? \]
\[ ? \longrightarrow \quad ? \]
Decoding Step

The red line indicates the right node has one edge.
Decoding Results

• Successful decoding corresponds to erasing the entire bipartite graph.
• Equivalently: graph has an empty 2-core.
• Using peeling analysis can determine when graph 2-core is empty with high probability.
  – Thresholds can be found by
    • Differential equation analysis
    • “And-Or” tree analysis
Tabulation Hashing

<table>
<thead>
<tr>
<th>Char 1</th>
<th>Char 2</th>
<th>Char 3</th>
<th>Char 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>101001010101</td>
<td>000011010101</td>
<td>111100111100</td>
</tr>
<tr>
<td>00000001</td>
<td>110100101010</td>
<td>101010101010</td>
<td>101101010101</td>
</tr>
<tr>
<td>00000010</td>
<td>001001011010</td>
<td>101010000000</td>
<td>000010110101</td>
</tr>
<tr>
<td>00000011</td>
<td>010100001100</td>
<td>001010100100</td>
<td>101010010111</td>
</tr>
<tr>
<td>00000100</td>
<td>010100101000</td>
<td>101010100111</td>
<td>000011001011</td>
</tr>
<tr>
<td>00000101</td>
<td>010101010000</td>
<td>000010010101</td>
<td>111010100001</td>
</tr>
<tr>
<td>00000110</td>
<td>110011001110</td>
<td>110010101110</td>
<td>000110101010</td>
</tr>
<tr>
<td>00000111</td>
<td>101010100001</td>
<td>110000011110</td>
<td>000110100000</td>
</tr>
</tbody>
</table>

Example: Hash 32 bit string into a 12 bit string using a table of 32*12 random bits.
## Tabulation Hashing

<table>
<thead>
<tr>
<th>Char 1</th>
<th>Char 2</th>
<th>Char 3</th>
<th>Char 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>101001010101</td>
<td>000011010101</td>
<td>111100111100</td>
</tr>
<tr>
<td>00000001</td>
<td>110100101010</td>
<td>101010101010</td>
<td>101101010101</td>
</tr>
<tr>
<td>00000010</td>
<td>001001011101</td>
<td>101010100000</td>
<td>000010110101</td>
</tr>
<tr>
<td>00000011</td>
<td>010100001100</td>
<td>001010100100</td>
<td>101010010111</td>
</tr>
<tr>
<td>00000100</td>
<td>010100101000</td>
<td>101010100111</td>
<td>000011001011</td>
</tr>
<tr>
<td>00000101</td>
<td>010101010000</td>
<td>000010010101</td>
<td>111010100001</td>
</tr>
<tr>
<td>00000110</td>
<td>110011001110</td>
<td>110010101110</td>
<td>000110101010</td>
</tr>
<tr>
<td>00000111</td>
<td>101010100001</td>
<td>110000011110</td>
<td>000011010000</td>
</tr>
</tbody>
</table>

Hash(00000000000000100000011100000001)
Tabulation Hashing

<table>
<thead>
<tr>
<th>Char 1</th>
<th>Char 2</th>
<th>Char 3</th>
<th>Char 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>101001010101</td>
<td>000011010101</td>
<td>111100111100</td>
</tr>
<tr>
<td>00000001</td>
<td>110100101010</td>
<td>101010101010</td>
<td>101101010101</td>
</tr>
<tr>
<td>00000010</td>
<td>001001011010</td>
<td>101010100000</td>
<td>000011001011</td>
</tr>
<tr>
<td>00000011</td>
<td>010100001100</td>
<td>001010100100</td>
<td>101010010111</td>
</tr>
<tr>
<td>00000100</td>
<td>010100101000</td>
<td>101010100111</td>
<td>000011001011</td>
</tr>
<tr>
<td>00000101</td>
<td>010101010000</td>
<td>000010010101</td>
<td>111010100001</td>
</tr>
<tr>
<td>00000110</td>
<td>110011001110</td>
<td>110010101110</td>
<td>000110101010</td>
</tr>
<tr>
<td>00000111</td>
<td>101010100001</td>
<td>111000011110</td>
<td>000011010000</td>
</tr>
</tbody>
</table>

....

Hash(000000000000000100000011100000001)
### Tabulation Hashing

<table>
<thead>
<tr>
<th>Char 1</th>
<th>Char 2</th>
<th>Char 3</th>
<th>Char 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>101001010101</td>
<td>000011010101</td>
<td>111100111100</td>
</tr>
<tr>
<td>00000001</td>
<td>110100101010</td>
<td>101010101010</td>
<td>101101010101</td>
</tr>
<tr>
<td>00000010</td>
<td>001001011010</td>
<td>101010100000</td>
<td>000011010000</td>
</tr>
<tr>
<td>00000011</td>
<td>010100001100</td>
<td>001010100100</td>
<td>101010010111</td>
</tr>
<tr>
<td>00000100</td>
<td>010101010000</td>
<td>101010010111</td>
<td>000011001011</td>
</tr>
<tr>
<td>00000101</td>
<td>010101010000</td>
<td>000010010101</td>
<td>111010100001</td>
</tr>
<tr>
<td>00000110</td>
<td>110011001110</td>
<td>110010101110</td>
<td>000111010101</td>
</tr>
<tr>
<td>00000111</td>
<td>101010100001</td>
<td>111000011110</td>
<td>000011010000</td>
</tr>
</tbody>
</table>

.....

**Hash(00000000000000100000011100000001)**

= 101001010101 xor 101010100000 xor 000011010000 xor 101010000111
Peeling and Tabulation Hashing

• Given a set of strings being hashed, we can peel as follows:
  – Find a string that in the set that uniquely maps to a specific (character value, position).
  – Remove that string, and continue.

• The peeled set has completely independent hashed values.
  – Because each one is xor’ed with its very own random table value.
Tabulation Hashing

• Result due to Patrascu and Thorup.
• Lemma: Suppose we hash \( n \leq m^{1-\text{eps}} \) keys of \( c \) chars into \( m \) bins, for some constant \( \text{eps} \). For any constant \( a \), all bins get less than \( d = ((1+a)/\text{eps})^c \) keys with probability \( \geq 1 - m^{-a} \).
  – Bootstrap this result to get Chernoff bounds for tabulation hashing.
Proof

• Let \( t = (1+a)/\epsilon \), so \( d = t^c \).
• Step 1: Every set of \( d \) elements has a peelable subset of size \( t \).
  – Pigeonhole principle says some character position is hit in \( d^{1/c} \) characters; otherwise fewer than \( (d^{1/c})^c = d \) total elements. Choose 1 for each character hit in that position.
• Step 2: Use this to bound maximum load.
  – Maximum load is \( d \) implies some \( t \)-element peelable set landed in the same bit. At most \( \binom{n}{t} \) such sets; \( m^{-(t-1)} \) probability each set lands in the same bin. Union bound gives heavily loaded bin with prob \( m^{-a} \).
End Survey

• Now, on to some new stuff.
Invertible Bloom Lookup Tables

Michael Goodrich
Michael Mitzenmacher
Bloom Filters, + Values + Listing

• Bloom filters useful for set membership.
• But they don’t allow (key,value) lookups.
  – Bloomier filters, extension to (key,value) pairs.
    • Also based on peeling methods.
• They also don’t allow you to reconstruct the elements in the set.
• Can we find something that does key-value pair lookups, and allows reconstruction?
  **Invertible Bloom Lookup Table (IBLT)**
Functionality

• IBLT operations
  – Insert (k,v)
  – Delete (k,v)
  – Get(k)
    • Returns value for k if one exists, null otherwise
    • Might return “not found” with some probability
  – ListEntries()
    • Lists all current key-value pairs
    • Succeeds as long as current load is not too high
      – Design threshold
Listing, Details

- Consider data streams that insert/delete a lot of pairs.
  - Flows through a router, people entering/leaving a building.
- We want listing not at all times, but at “reasonable” or “off-peak” times, when the current working set size is bounded.
  - If we do all the N insertions, then all the N-M deletions, and want a list at the end, we want...
- Data structure size should be proportional to listing size, not maximum size.
  - Proportional to M, not to N!
  - Proportional to size you want to be able to list, not number of pairs your system has to handle.
Sample Applications

• Network flow tracking
  – Track flows on insertions/deletions
  – Possible to list flows at any time – as long as the network load is not too high
    • If too high, wait till it gets lower again
  – Can also do flow lookups (with small failure probability)

• Oblivious table selection

• Database/Set Reconciliation
  – Alice sends Bob an IBLT of her data
  – Bob deletes his data
  – IBLT difference determines set difference
Possible Scenarios

• A nice system
  – Each key has (at most) 1 value
  – Delete only items that are inserted
• A less nice system
  – Keys can have multiple values
  – Deletions might happen for keys not inserted, or for the wrong value
• A further less nice system
  – Key-value pairs might be duplicated
The Nice System

(k,v) pair hashed to j cells

Get : If Count = 0 in any cell, return null
Else, if Count = 1 in any cell and KeySum = key,
return ValueSum
Else, return Not Found

Insert : Increase Count,
Update KeySum and ValueSum

Delete : Decrease Count,
Update KeySum and ValueSum
Get Performance

• Bloom filter style analysis

• Let $m =$ number of cells, $n =$ number key-value pairs, $j =$ number of hash functions

• Probability a Get for a key $k$ in the system returns “not found” is

$$
\left(1 - e^{-jn/m}\right)^j
$$

• Probability a Get for a key $k$ not in the system returns “not found is”

$$
\left(1 - e^{-jn/m} - \frac{jn}{m} e^{-jn/m}\right)^j
$$
The Nice System : Listing

• While some cell has a count of 1:
  – Set \((k,v) = (\text{KeySum}, \text{ValueSum})\) of that cell
  – Output \((k,v)\)
  – Call Delete\((k,v)\) on the IBLT
Listing Example
The Nice System : Listing

• While some cell has a count of 1:
  – Set \((k,v) = (KeySum,ValueSum)\) of that cell
  – Output \((k,v)\)
  – Call Delete\((k,v)\) on the IBLT

• Peeling Process.
• This is the same process used to find the 2-core of a random hypergraph.
• Same process used to decode families of low-density parity-check codes.
Listing Performance

• Results on random peeling processes
• Thresholds for complete recovery depend on number of hash functions

<table>
<thead>
<tr>
<th>J</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>m/n</td>
<td>1.222</td>
<td>1.295</td>
<td>1.425</td>
<td>1.570</td>
</tr>
</tbody>
</table>

• Interesting possibility: use “irregular” IBLTs
  – Different numbers of hash functions for different keys
  – Same idea used in LDPC codes
Fault Tolerance

• Extraneous deletions
  – Now a count of 1 does not mean 1 key in the cell.
    • Might have two inserted keys + one extraneous deletion.
  – Need an additional check: hash the keys, sum into HashKeySum.
    – If count is 1, and the hash of the KeySum = HashKeySum, then 1 key in the cell.

• What about a count of -1?
  – If count is -1, and the hash of -KeySum = -HashKeySum, then 1 key in the cell.
Fault Tolerance

• Keys with multiple values
  – Need another additional check; HashKeySum and HashValueSum.

• Multiply-valued keys “poison” a cell.
  – The cell is unusable; it will never have a count of 1.

• Small numbers of poisoned cells have minimal effect.
  – Usually, all other keys can still be listed.
  – If not, number of unrecovered keys is usually small, like 1.
Biff Codes
Using IBLTs for Codes

George Varghese
Michael Mitzenmacher
Result

• Fast and simple low-density parity-check code variant.
  – For correcting errors on q-ary channel, for reasonable-sized q.

• All expressed in terms of “hashing”
  – No graphs, matrices needed for coding.

• Builds on intrinsic connection between set reconciliation and coding.
  – Worth greater exploration.

• Builds on intrinsic connection between LDPC codes and various hashing methods.
  – Previously used for e.g. thresholds on cuckoo hashing.
Reminder:
Invertible Bloom Lookup Tables

• A Bloom filter like structure for storing key-value pairs (k,v).

• Functionality:
  – Insert (k,v)
  – Delete (k,v)
  – Get(k)
    • Returns value for k if one exists, null otherwise
    • Might return “not found” with some probability
  – ListEntries()
    • Lists all current key-value pairs
    • Succeeds as long as current load is not too high
      – Design threshold
The Basic IBLT Framework

(k,v) pair hashed to j cells

Insert: Increase Count, Update KeySum and ValueSum
Delete: Decrease Count, Update KeySum and ValueSum

Notes: KeySum and ValueSum can be XORs
Counts can be negative! (Deletions without insertion.)
In fact, in our application, counts are not needed!
Set Reconciliation Problem

- Alice and Bob each hold a set of keys, with a large overlap.
  - Example: Alice is your smartphone phone book, Bob is your desktop phone book, and new entries or changes need to be synched.
- Want one/both parties to learn the set difference.
- Goal: communication is proportional to the size of the difference.
- IBLTs yield an effective solution for set reconciliation. (Used in code construction...)
Biff Code Framework

• Alice has message $x_1, x_2, \ldots, x_n$.
  – Creates IBLT with ordered pairs as keys, $(x_1,1), (x_2,2), \ldots (x_n,n)$.
  – Values for the key are a checksum hash of the key.
  – Alice’s “set” is $n$ ordered pairs.
  – Sends message values and IBLT.

• Bob receives $y_1, y_2, \ldots y_n$.
  – Bob’s set has ordered pairs $(y_1,1), (y_2,2), \ldots (y_n,n)$.
  – Bob uses IBLT to perform set reconciliation on the two sets.
Reconciliation/Decoding

• For now, assume no errors in IBLT sent by Alice.
• Bob deletes his key-value pairs from the IBLT.
• What remains in the IBLT is the set difference, corresponding to symbol errors.
• Bob lists elements of the IBLT to find the errors.
Decoding Process

• Suppose a cell has one pair in it.
• Then the checksum should match with the key value.
  – And, if more than one pair, checksum should not match with the key value. Choose checksum length so no false matches with good probability.
• If checksum matches, recover the element, and delete it from the IBLT.
• Continue until all pairs recovered.
Peeling Process

Keys

Hash Table

\[
\begin{align*}
\text{Keys} & : a, ?, c, ?, f, ? \\
\text{Hash Table} & : b, b \oplus g, e \oplus g, b \oplus e, e \oplus g
\end{align*}
\]
Analysis

• Analysis follows standard approaches for LDPC codes.
  – E.g., differential equation, fluid limit analysis.
  – Chernoff-like bounds on behavior.

• Overheads
  – Set difference size = 2x number of errors.
    • Could we get rid of factor of 2?
  – Decoding structure overhead.
    • Best not to use “regular graph” = same number of hash functions per item; but simpler to do so.
Fault Tolerance

- What about errors in IBLT cells?
  - The cell is “bad”, can’t be used for recovery.
  - If the checksum works, low probability of decoding error.
  - Enough bad IBLT cells will harm decoding.
    - Most likely scenario: one key-value pair has all of its IBLT cells go bad; it cannot be recovered.
    - So most likely error: 1 unrecovered value (or small number).

- Various remedies possible.
  - Small additional error-correction in original message.
  - Recursively protect IBLT cells with a code.
Simple Code

Code is essentially a lot of hashing, XORing of values.

- **ENCODE**
  
  \[
  \begin{align*}
  \text{for } i = 1 \ldots n \text{ do} \\
  &\text{for } j = 1 \ldots k \text{ do} \\
  &T_j[h_j((x_i, i))].\text{keySum} \triangleq (x_i, i). \\
  &T_j[h_j((x_i, i))].\text{valueSum} \triangleq \text{Check}((x_i, i)).
  \end{align*}
  \]

- **DECODE**
  
  \[
  \begin{align*}
  \text{for } i = 1 \ldots n \text{ do} \\
  &\text{for } j = 1 \ldots k \text{ do} \\
  &T_j[h_j((y_i, i))].\text{keySum} \triangleq (y_i, i). \\
  &T_j[h_j((y_i, i))].\text{valueSum} \triangleq \text{Check}((y_i, i)).
  \end{align*}
  \]

  while \( \exists \ a, j \) with \((T_j[a].\text{keySum} \neq 0)\) and \((T_j[a].\text{valueSum} == \text{Check}(T_j[a].\text{keySum}))\) do

  \[
  (z, i) = T_j[a].\text{keySum}
  \]

  if \( z \neq y_i \) then
  set \( y_i \) to \( z \) when decoding terminates

  \[
  \begin{align*}
  \text{for } j = 1 \ldots k \text{ do} \\
  &T_j[h_j((z, i))].\text{keySum} \triangleq (z, i). \\
  &T_j[h_j((z, i))].\text{valueSum} \triangleq \text{Check}((z, i)).
  \end{align*}
  \]
Experimental Results

• Test implementation.
  – 1 million 20-bit symbols.
    • 20 bits also describes location, used for checksum.
    • All computations are 64 bits.
  – IBLTs of 30000 cells.
  – 10000 symbol errors (1% rate)
  – 600 IBLT errors (2% rate).
Experimental Results

• Parameters chosen so we expect rare but noticeable failures with 4 hash functions.
  – All IBLT cells for some symbol in error in approximately $1.6 \times 10^{-3}$ trials.
• But less so with 5 hash functions.
  – Failure once in approximately $3.2 \times 10^{-5}$ trials.
• 16 failures in 1000 trials for 4 hash functions, none for 5 hash functions.
  – Failures are all 1 unrecovered element!
• Experiments match analysis.
Experimental Results: Timing

• Less than 0.1 seconds per decoding.
• Most of the time is simply putting elements into the hash table.
  – 4 hash functions: 0.0561 seconds on average to load data into table, 0.0069 for subsequent decoding.
  – 5 hash functions: 0.0651 seconds on average to load data into table, 0.0078 for subsequent decoding.
• Optimizations, parallelizations possible.
Conclusions for Biff Codes

• Biff Codes are extremely simple LDPC-style codes for symbol-level errors.
  – Simple code.
  – Very fast.
  – Some cost in overhead.

• Expectation: will prove useful for large-data scenarios.

• Didactic value: an easy way to introduce LDPC concepts.
Peeling

- Peeling method is a useful approach when trying to analyze random graph processes.
- Many problems can be put in this framework.
  - Matchings
  - Codes
  - Hash tables
  - Bloom filter structures
- Designing processes to use peeling leads to quick algorithms.
  - Simple, time-efficient; often space tradeoffs.
- What data structures/algorithms can we put in the peeling framework?