Advanced Data Structures

Lecture 11: Minimal Perfect Hashing

Florian Kurpicz
Recap: Retroactive Data Structures

**Operations**
- \( \text{INSERT}(t, \text{operation}) \): insert operation at time \( t \)
- \( \text{DELETE}(t) \): delete operation at time \( t \)
- \( \text{QUERY}(t, \text{query}) \): ask query at time \( t \)

For a priority queue updates are
- insert
- delete-min

Time is integer ⌦ for simplicity otherwise use order-maintenance data structure

**Definition: Partial Retroactivity**
QUERY is only allowed for \( t = \infty \) now

**Definition: Full Retroactivity**
QUERY is allowed at any time \( t \)

**Definition: Nonoblivious Retroactivity**
INSERT, DELETE, and QUERY at any time \( t \) but also identify changed QUERY results
Hashing (1/2)

- **Definition: Totally Random**
  - $\Pr[h(x) = t] = 1/m$
  - Independent of $h(y)$ for all $x \neq y \in U$
  - Requires $\Theta(u \log m)$ bits of space to store too big

- **Definition: Universal**
  - Choose $h$ from family $H$ with
    $\Pr_{h \in H}[h(x) = h(y)] = O(1/m)$ for all $x \neq y \in U$.
  - Family is small to enable efficient encoding

- $h: \{0, \ldots, u-1\} \rightarrow \{0, \ldots, m-1\}$
- $n$ objects
- From universe $U = \{0, \ldots, u-1\}$
- Hash table of size $m \approx m$ close to $n$
- $m \ll u$

- $h(x) = (ax \mod u) \mod m$ for $0 < a < p$ and $p$ being prime $> u$
- $h(x) = ax \gg (\log u - \log m)$ for $m, u$ being powers of two

Why is this family easier to store?
Definition: $k$-wise Independent

- choose $h$ from family $H$ with
  $\Pr[h(x_1) = t_1 & \ldots & h(x_k) = t_k] = O(1/m^k)$ for distinct $x_1, \ldots, x_k \in U$
- implies universal
- $h(x) = ((\sum_{i=0}^{k-1} a_i x^i) \mod p) \mod m$ for $0 \leq a_i < p$ and $0 < a_{k-1} < p$
- pairwise ($k = 2$) independence is stronger than universal
- $h(x) = ((ax + b) \mod u) \mod m$

Definition: Simple Tabulation Hashing

- view $x$ as vector $x_1, \ldots, x_c$ of characters
- totally random hash table $T_i$ for each character
- $h(x) = T_1(x_1) \oplus \ldots \oplus T_c(x_c)$
- Why can we use totally random hash tables?
- $O(cu^{1/c})$ space
- $O(c)$ time to compute
- 3-wise independent
Minimal Perfect Hashing

**Definition: Perfect Hash Function**
- injective hash function
- maps \( n \) objects to \( m \) slots

- lower space bound for \( m = (1 + \epsilon)n \) is
  \[
  \log e - \epsilon \log \frac{1 + \epsilon}{\epsilon}
  \]
- for \( m \) close to \( n \) there are likely collisions

**Definition: Minimal Perfect Hash Function**
- bijective hash function
- maps \( n \) objects to \( m = n \) slots
- \( h: N \rightarrow [0, n) \)

- lower space bound as for PHF with \( \epsilon = 0 \):
  \[
  \log e \approx 1.44
  \]
- no collisions

- can we make PHF to MPHF?
for each object calculate three potential slots \((h_0, h_1, \text{ and } h_2)\)

for each slot that contains only one object, remove the object from all its other slots

one slot per object

if that does not work use other hash functions

use rank data structure to map slots to \([0, n)\)

how to check if hash function works

interpret each slot as node in a hypergraph

objects are edges

if graph is peelable, we have a feasible mapping

Definition: Peelable

A hypergraph is peelable, if it is possible to obtain a graph without edges by iteratively taking away edges that contain a node with degree 1

1.95 bits per object when \(m = 1.23 \times n\)
Compress, Hash, and Displace [BBD09a]

- partition keys into buckets
- set $m = (1 + \epsilon)n \approx 1.01n$
- sort partitions by size
- starting with largest bucket, find universal hash function mapping all keys to empty slots
- if key mapped to non-empty slot, try next hash function
- for each bucket store universal hash function
- use rank data structure to map slots to $[0, n)$

- can be used as PHF
- there are a lot of tricks w.r.t. bucket sizes and size distributions
- requires around 2.05 bits per object

- example on the board 📚
RecSplit Overview [EGV20a]

- partition keys into buckets of size $b$
- for each bucket compute splitting trees
- split keys into smaller sets
- stop when sets have size $\ell$

- upper aggregation levels have fanout 2
- lower two aggregation levels have fanout
  - $\max\{2, \lceil 0.35\ell + 0.55 \rceil \}$
  - $\max\{2, \lceil 0.21\ell + 0.9 \rceil \}$

- last level is leaf level
- find bijections
RecSplit Splitting Tree

- tree structure is well defined
- store information for each node in preorder
- store hash function for each splitter
- encode function using Golomb-Rice

- encodings of splitting trees stored in one bit vector
- use Elias-Fano to store
  - size of buckets
  - starting position of bucket in bit vector
**Definition: Golomb Code**

Given an integer $x > 0$ and a constant $b > 0$, the Golomb code consists of

- $q = \lfloor \frac{x}{b} \rfloor$
- $r = x - qb = x \% b$
- $c = \lceil \log b \rceil$

with

$$(x)_{\text{Gol}}(b) = (q)_1(r)_2$$

where $(r)_2$ depends on its size

- $r < 2^{\lfloor \log b \rfloor - 1}$: $r$ requires $\lfloor \log b \rfloor$ bits and starts with a 0
- $r \geq 2^{\lfloor \log b \rfloor - 1}$: $r$ requires $\lfloor \log b \rfloor$ bits and starts with a 1 and it encodes $r - 2^{\lfloor \log b \rfloor - 1}$

- $b$ has to be fixed for all codes
- still variable length
- Golomb-Rice is special case where $r$ is power of two

- for $b = 5$, there are 4 remainders: 00, 01, 100, 101, and 110
- $2^{\lfloor \log 5 \rfloor - 1} = 2$
- 0, 1 < 2: 00 and 01 require 2 bits
- 2, 3, 4 ≥ 2: require 3 bits and encode 0, 1, 2 starting with 1
Comparison of Codes (1/2)

The diagram compares the size of various encoding schemes for values ranging from 0 to 50. The schemes include:

- **unary**
- **ternary**
- **Fibonacci**
- **Elias-γ**
- **Elias-δ**
- **Golomb (b = 5)**
- **Golomb (b = 10^6)**

The x-axis represents the value, and the y-axis represents the size of the encoding.
Comparison of Codes (2/2)

- unary
- ternary
- Fibonacci
- Elias-γ
- Elias-δ
- Golomb ($b = 5$)
- Golomb ($b = 10^6$)
RecSplit Leaves

- find perfect hash function for keys in leaves
- test hash functions brute force
- use hash value modulo $\ell$
- set bit in “bit vector” of length $\ell$
- all bits set indicates bijection

Input objects

Bucket 0

Bucket 1

Bucket 2
RecSplit Queries

- find bucket
- follow splitting tree
- accumulate number of objects to the left
- use bijection in leaf
- result is sum of
  - objects in previous buckets
  - objects to the left in splitting tree
  - value of bijection
Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions”. In: CoRR abs/2212.09562 (2022)

- results to be presented at ESA’23
- based on a Domink Bez’ Master’s thesis
randomly distribute objects in leaf in two sets $A$ and $B$
hash objects in both set
two "bit vectors": cyclic shift one until all bits are set when 0Red
store hash function and rotation

Lemma: Rotation Fitting

Let $|A| = A$, $|B| = B$, and $P(R)$ be the probability of finding a bijection using rotation fitting. Let $P(B)$ denote the probability of finding a bijection using RecSplit's brute force strategy. Then, $P(R) \rightarrow mP(B)$ for $m \rightarrow \infty$. 

![Graph showing expected space overhead and higher probability vs. m]
Rotation Fitting (2/3)

Proof (Sketch)

- consider number of different injective functions under cyclic shifts
- bit vector of length $m$ with $B$ set bits
- total number of equivalence classes under rotation is $\frac{1}{m} \sum_{d \text{ divides } \gcd(A, B)} \phi(d) \left( \frac{m}{d} \right)$
- probability of the event $I$ that there is a rotation has the $m$ least significant bits set is

$$\Pr(I) \geq \frac{1}{m} \sum_{d \text{ divides } \gcd(A, B)} \phi\left( \frac{m}{d} \right) \phi\left( \frac{B}{d} \right)$$

- $\phi(i) = |\{j \leq i : \gcd(i, j) = 1\}|$ is Euler's totient function

Proof (Sketch, cnt.)

- determine the probability $\Pr(R)$ using the events
  - $A$: popcount(a)=$A$
  - $B$: popcount(b)=$B$
  - $B$: found bijection using brute-force
Rotation Fitting (3/3)

Proof (Sketch, ctn.)

\[
P(R) = P(A)P(B)P(I) \\
\geq \frac{m!}{(m-A)!m^A} \cdot \frac{m!}{(m-B)!m^B} \cdot P(I) = \frac{m!}{m^m} \cdot \frac{m!}{A!B!} \cdot P(I) = P(B) \cdot \frac{m!}{A!B!} \cdot P(I) \\
\geq P(B) \cdot \frac{m!}{A!B!} \cdot m \cdot \frac{1}{\sum d \mid gcd(A,B) \phi(d) (\frac{m/d}{b/d})} = P(B) \cdot m \cdot \frac{m!}{m! + (A!B!) \sum d \mid gcd(A,B), d \neq 1 \phi(d) (\frac{m/d}{b/d})} \\
= P(B) \cdot m \cdot \frac{1}{1 + \sum d \mid gcd(A,B), d \neq 1 \phi(d) \frac{(m/d)!A!B!}{m! (A/d)! (B/d)!}} \\
\sim P(B) \cdot m \cdot \frac{1}{1 + \sum d \mid gcd(A,B), d \neq 1 \phi(d) \sqrt{d} \left(\frac{A^A - A^d}{B^B - B^d} \right)m^{m-m/d}} \\
\rightarrow P(B) \cdot m \text{ for } m \rightarrow \infty
\]
Parallel RecSplit on the GPU

Computing on the GPU
- several streaming multiprocessors (SMs)
- each SM contains many arithmetic logic units (ALUs)
- several threads operate in lock-step (warp)
- to hide latencies, each SM is oversubscribed with more threads than ALUs
- in CUDA, kernels are functions that can be executed on the GPU
- a kernel is executed on a grid of thread blocks

- use GPU to determine splitting and bijections
Experimental Evaluation

- Intel i7 11700 processor with 8 cores (16 hardware threads (HT)), base clock: 2.5 GHz
- AVX-512.
- Ubuntu 22.04 with Linux 5.15.0
- NVIDIA RTX 3090 GPU

- AMD EPYC 7702P processor with 64 cores (128 hardware threads), base clock: 2.0 GHz
- AVX2
- Ubuntu 20.04 with Linux 5.4.0

- GNU C++ compiler v.11.2.0 (-03 -march=native)
Rotation Fitting

![Graph showing the performance of Brute force and Rotation fitting methods.](image)

- **Brute force**
- **Rotation fitting**

**Bits per object** vs **Objects/second**

**Speedup**

- Brute force and Rotation fitting comparison.
### Overview Results

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Method</th>
<th>Bijections</th>
<th>Threads</th>
<th>B/Obj</th>
<th>Constr.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell = 16, b = 2000$</td>
<td>RecSplit [EGV20b]</td>
<td>Brute force</td>
<td>1</td>
<td>1.560</td>
<td>1175.4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RecSplit</td>
<td>Brute force</td>
<td>16</td>
<td>1.560</td>
<td>206.5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>1</td>
<td>1.560</td>
<td>138.0</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>16</td>
<td>1.560</td>
<td>27.9</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Brute force</td>
<td>GPU</td>
<td>1.560</td>
<td>1.8</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Rotation fitting</td>
<td>GPU</td>
<td>1.560</td>
<td>1.0</td>
<td>1173</td>
</tr>
<tr>
<td>$\ell = 18, b = 50$</td>
<td>RecSplit [EGV20b]</td>
<td>Brute force</td>
<td>1</td>
<td>1.707</td>
<td>2942.9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RecSplit</td>
<td>Brute force</td>
<td>16</td>
<td>1.713</td>
<td>504.0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>1</td>
<td>1.709</td>
<td>58.3</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>16</td>
<td>1.708</td>
<td>12.3</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Brute force</td>
<td>GPU</td>
<td>1.708</td>
<td>5.2</td>
<td>564</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Rotation fitting</td>
<td>GPU</td>
<td>1.709</td>
<td>0.5</td>
<td>5438</td>
</tr>
<tr>
<td>$\ell = 24, b = 2000$</td>
<td>GPURecSplit</td>
<td>Brute force</td>
<td>GPU</td>
<td>1.496</td>
<td>2300.9</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Rotation fitting</td>
<td>GPU</td>
<td>1.496</td>
<td>467.9</td>
<td>—</td>
</tr>
</tbody>
</table>
Comparison with Competitors

8-core Intel
1 Thread

8-core Intel
16 Threads

64-core AMD
1 Thread

64-core AMD
128 Threads

Throughput (MObjects/s)

Bits/Object

BBHash [Lim+17]  CHD [BBD09b]  PTHash [PT21b]  PTHash-HEM [PT21a]
RecSplit [EGV20b]  SIMDRecSplit  SicHash [LSW23]
Conclusion and Outlook

This Lecture
- minimal perfect hash functions

Next Lecture
- dynamic data structures

Advanced Data Structures

- retroactive PQ
- String B-tree
- SA & LCP
- Successor
- CSA
- RMQ
- static/dynamic BV
- static/dynamic succ. trees
- range min-max tree
- succ. graphs
Bibliography I


Bibliography II


Bibliography III

