Data Structures for Range Minimum Queries in Multidimensional Arrays

Hao Yuan   Mikhail J. Atallah

Purdue University

January 17, 2010
OUTLINE

- Introduction
- Results
  - Overview
  - Details
    - Step 1: Comparison-Efficient Data Structures
    - Step 2: Random Access Machine Implementation
- Future Work
Definitions

Given a $d$-dimensional array $A$ with $N$ entries, a Range Minimum Query (RMQ) asks the minimum element in the query range $q = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$, i.e.,

$$\text{RMQ}(A, q) = \min A[q] = \min_{(k_1, \ldots, k_d) \in q} A[k_1, \ldots, k_d].$$
Applications

- **String Pattern Matching**: 1D RMQ and its related Least Common Ancestor (LCA) problems are fundamental building blocks in suffix trees/arrays.
- **Computational Biology**: Finding min/max number in an alignment tableau (genome sequence analysis).
- **Image Processing**: Finding the lightest/darkest point in a range (Dilate/Erode Filter).
- **Databases**: Range Min/Max Query in OLAP Data Cube.

**Example**

Select the highest paid employee whose age is between 30 and 40 and joined the company during the period between 1995 and 2005.
1D Range Minimum Query

- Linear Reduction to *Least Common Ancestor* (LCA) Problem
  [Gabow, Bentley and Tarjan 1984]

- LCA: $O(N)$ Preprocessing, $O(1)$ Querying
  [Harel and Tarjan 1984]

- RMQ & LCA: Much Studied
  (Parallelization, Simplification, Distributed Algorithms, etc)
  [Schieber and Vishkin 1988]
  [Bender and Farach-Colton 2000]
  [Alstrup et al. 2002]
Related to the semi-group sum problem (MIN is a semi-group operator)
Data Structures: $O(M)$ preprocessing time and space ($M \geq N$),
$O(\alpha(M, N))$ querying time
- One Dimensional: [Yao 1982], [Alon and Schieber 1987]
- Multidimensional (fixed $d$): [Chazelle and Rosenberg 1989]
**Multidimensional RMQ**

**Unit-Cost RAM Model:**
O(1) cost for: Read/Write Memory, +, −, *, /, <<, >>

**Comparison-Based:** Array entries can only be compared

**Table:** Results for \(d\)-dimensional RMQ (\(d\) is fixed). The \(O(\cdot)\) is omitted.

<table>
<thead>
<tr>
<th></th>
<th>Preprocess Time</th>
<th>Space</th>
<th>Querying Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gabow et al. 1984</td>
<td>(N \log^{d-1} N)</td>
<td>(N \log^{d-1} N)</td>
<td>(\log^{d-1} N)</td>
</tr>
<tr>
<td>Chazelle and Rosenberg 1989</td>
<td>(M)</td>
<td>(M)</td>
<td>(\alpha^d(M, N))</td>
</tr>
<tr>
<td>Poon 2003</td>
<td>(N(\log^* N)^d)</td>
<td>(N)</td>
<td>1</td>
</tr>
<tr>
<td>Amir et al. 2007 ((d = 2))</td>
<td>(N \log^{[k+1]} N)</td>
<td>(kN)</td>
<td>1</td>
</tr>
<tr>
<td><strong>Our result</strong></td>
<td>(N)</td>
<td>(N)</td>
<td>1</td>
</tr>
</tbody>
</table>
Overview

General Approach

- Design Comparison-Efficient Algorithm:
  Only comparisons between input array entries are counted

- Implement the Algorithm in RAM:
  All the computations are counted

Example: Minimum Spanning Tree Verification
[Komlós 1984] [Dixon, Rauch and Tarjan 1992]
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Overview

Following the general approach:

- **Comparison-Efficient Data Structures**
  - **New 1D RMQ**
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Overview

Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Overview

Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
    - Solve big size query by well-known algorithms
    - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
Following the general approach:

- **Comparison-Efficient Data Structures**
  - New 1D RMQ
    - Preliminary: $O(N \log N)$-comparison preprocessing and 1-comparison querying
    - Speedup the preprocessing to $O(N)$ comparisons
  - New data structure generalizes to two or higher dimensional cases
    - Preprocessing: $O(N)$ comparisons
    - Querying: $O(1)$ comparisons

- **RAM Implementations**
  - Micro blocks of size $\epsilon \log N$
  - Solve big size query by well-known algorithms
  - Solve small size query by table lookup
If only count comparisons:

- **2D RMQ Lower Bound:** [Demaine, Landau and Weimann, 2009]
  If *NO COMPARISON* is allowed at the query stage, then $\Omega(N \log N)$ comparisons preprocessing is required

- **Our Result:** $O(2.89^d(d + 1)!N)$ comparisons preprocessing, $2^d - 1$ comparisons querying
Canonical Ranges

\[ \text{CR}(x) = [5, 8] \]
For each \( p \in CR(x) \), define

\[
\begin{align*}
\text{LeftMin}(x, p) &= \min_{k \in CR(x) \text{ and } k \leq p} A[k] \\
\text{RightMin}(x, p) &= \min_{k \in CR(x) \text{ and } k \geq p} A[k]
\end{align*}
\]

\[
\text{LeftMin}(x, 7) = \min\{ A[5], A[6], A[7] \} \\
\text{RightMin}(x, 7) = \min\{ A[7], A[8] \}
\]
min A[6..14]

LCA(6, 14)

RightMin(x, 6)  LeftMin(y, 14)
Pre-Computations

- Naïve Algorithm: $O(N \log N)$ Comparisons
- Faster Approach
  - Sort the canonical ranges by their lengths
  - Compute the LeftMin and RightMin entries for canonical ranges in the sorted order
    - For length-one canonical range $CR(w)$,
      $LeftMin(w, p) = RightMin(w, p) = A[p]$ ($p \in CR(w)$)
    - For a canonical range $CR(w)$ covering more than one position, compute the LeftMin and RightMin arrays in $O(\log |CR(w)|)$ time (instead of $O(|CR(w)|)$)
Case 1: $p \in CR(x)$

\[
\text{LeftMin}(w, p) = \text{LeftMin}(x, p)
\]
Case 2: $p \in CR(y)$

$$\text{LeftMin}(w, p) = \min \left\{ \min CR(x), \text{LeftMin}(y, p) \right\}$$
Case 2: \( p \in CR(y) \)

\[
\text{LeftMin}(w, p) = \min \{ \min CR(x), \quad \text{LeftMin}(y, p) \}
\]

Monotonicity (Non-Increasing): \( \text{LeftMin}(y, p) \geq \text{LeftMin}(y, p + 1) \)

Binary Search!
Example

\[
\begin{align*}
\text{min } CR(x) &= 40 \\
\text{LeftMin}(y, \ldots) &= 90, 70, 50, 20, 10 \\
\text{LeftMin}(w, \ldots) &= 40, 40, 40, 20, 10
\end{align*}
\]
$T(n)$: the number of comparisons to compute the LeftMin and RightMin entries for canonical ranges whose size is at most $n$

\[
T(1) = 0 \\
T(n) = 2T \left( \frac{n}{2} \right) + O(\log n) \quad \text{for } n \geq 2
\]

We have the preprocessing comparison complexity

\[
T(n) = O(n),
\]

and need to do 1 comparison at the query stage.
2D Canonical Range: Cartesian Product of Two 1D Canonical Ranges
For each 2D canonical range $r$ and a point $p \in r$, compute the 4 "Dominance Min" array entries

- TopLeftMin($r, p$)
- TopRightMin($r, p$)
- BotLeftMin($r, p$)
- BotRightMin($r, p$)
For any query range $q$, we can always divide it into 4 parts, which are all pre-computed.
Efficient Pre-Computations

- For any canonical range \( r \), cut the middle of its longer side to obtain two smaller canonical ranges \( r_1 \) and \( r_2 \)
- Do binary search row by row (or column by column)
- \( O(N) \) comparisons for 2D preprocessing
- Generalize to any fixed dimension \( d \):
  - Preprocess: \( O(2.89^d(d + 1)!N) \) comparisons
  - Query: \( 2^d - 1 \) comparisons
Efficient Pre-Computations

- For any canonical range $r$, cut the middle of its longer side to obtain two smaller canonical ranges $r_1$ and $r_2$
- Do binary search row by row (or column by column)
- $O(N)$ comparisons for 2D preprocessing
- Generalize to any fixed dimension $d$:
  - Preprocess: $O(2.89^d(d + 1)!N)$ comparisons
  - Query: $2^d - 1$ comparisons
For any canonical range \( r \), cut the middle of its longer side to obtain two smaller canonical ranges \( r_1 \) and \( r_2 \).

- Do binary search row by row (or column by column).
- \( O(N) \) comparisons for 2D preprocessing.
- Generalize to any fixed dimension \( d \):
  - Preprocess: \( O(2.89^d(d + 1)!N) \) comparisons
  - Query: \( 2^d - 1 \) comparisons

**BotLeftMin**

```
Binary Search
r1 r2
  p
```

\( BotLeftMin(r, p) \)
Efficient Pre-Computations

- For any canonical range $r$, cut the middle of its longer side to obtain two smaller canonical ranges $r_1$ and $r_2$
- Do binary search row by row (or column by column)
- $O(N)$ comparisons for 2D preprocessing
- Generalize to any fixed dimension $d$:
  - Preprocess: $O(2.89^d(d + 1)!N)$ comparisons
  - Query: $2^d - 1$ comparisons

![Binary Search Diagram](image-url)
Divide the array into micro blocks of size $\epsilon \log N$

Each block is a $d$-dimensional cube, with side length $(\epsilon \log N)^{\frac{1}{d}}$

For example in 2D, make each block $\sqrt{\epsilon \log N}$ by $\sqrt{\epsilon \log N}$
For query that crosses the border of any micro block: there exists \( O(N) \)-time preprocessing and constant-time querying data structures to solve it, using dimension reductions and the help of the data structures in [Yao 1982] [Chazelle and Rosenberg 1989]
Overview of RAM Implementations

For query that is complete within a micro block, use table lookup technique (Four Russian’s Trick) to get the locations of at most $2^d$ candidates to compare at the querying stage.

Based on our linear-comparison preprocessing data structure.
Key Idea: If two micro blocks have the same type, then they should share the same data structures

- Type of a micro block: Comparison results (true/false sequence) of the linear-comparison preprocessing algorithm
- Assume $c\epsilon \log N$ comparisons to preprocess a block: at most $2^{c\epsilon \log N} = N^{c\epsilon}$ possible types
  - Choose $\epsilon < \frac{1}{c}$, then there are only a sublinear number of types:
    $$N^{c\epsilon \text{polylog}(\epsilon \log N)} = o(N)$$
- Recognizing the types for all micro blocks in linear time: Build a linear-depth decision tree according to the linear-comparison preprocessing algorithm
Our unit-cost RAM data structure

- Preprocess in $O(2.89^d(d + 1)!N)$ time and $(2^d d!N)$ space
- Query in $O(3^d)$ time
Future Work:

- Extend the lower bound of [Demaine, Landau and Weimann, 2009]
  - If at most $t$ comparisons are allowed at the querying stage, find the lower bound for the number of comparisons required to preprocess the input
- Dynamic Updates [Poon, 2003]
- Extend our results to the external memory model
Future Work:

- Extend the lower bound of [Demaine, Landau and Weimann, 2009]
  - If at most $t$ comparisons are allowed at the querying stage, find the lower bound for the number of comparisons required to preprocess the input.
- Dynamic Updates [Poon, 2003]
- Extend our results to the external memory model
Future Work:

- Extend the lower bound of [Demaine, Landau and Weimann, 2009]
  - If at most $t$ comparisons are allowed at the querying stage, find the lower bound for the number of comparisons required to preprocess the input

- Dynamic Updates [Poon, 2003]
  - Extend our results to the external memory model
**Future Work**

Future Work:

- Extend the lower bound of [Demaine, Landau and Weimann, 2009]
  - If at most $t$ comparisons are allowed at the querying stage, find the lower bound for the number of comparisons required to preprocess the input

- Dynamic Updates [Poon, 2003]

- Extend our results to the external memory model
Thank You!