Algorithms for Evolving Data Sets

Mohammad Mahdian

Google Research

Based on joint work with
Aris Anagnostopoulos, Bahman Bahmani, Ravi Kumar, Eli Upfal, and Fabio Vandin
Algorithm Design Paradigms

- **Traditional paradigm:**
  - stationary data set
  - algorithm has unrestricted access to data

- **Alternative paradigms:**
  - **Online algorithms**
    - Must make irrevocable decisions as data arrives
  - **Streaming algorithms**
    - Not enough space to store entire data set
  - **Sublinear time algorithms**
    - Not enough time to read entire data set
  - **Algorithmic game theory, …**
    - Feedback loop: choice of algo influences data
Evolving data: motivation

- Often data is a snapshot of the “nature”.
- The nature changes over time.
- Need to keep up with such changes by constantly observing the nature and adjusting the solution based on new observations.

Example:
- Computing PageRank, or other computations on the web graph
- Polling public opinion
- Finding paths to route traffic on a network
In this talk

- Define a general model for algorithm design on “evolving data”.
- Argue that the model is *practically useful* and *mathematically interesting* through three examples:
  - Sorting evolving data (ICALP 2009)
  - Basic graph algorithms (ITCS 2012)
  - PageRank computation (KDD 2012)
General Model

- At time $t$, real input $U_t$
- Need $v_t = f(U_t)$
- Input changes slowly stochastically (or adversarially):
  $$d(U_{t+1}, U_t) = \text{small}$$
- Algorithm can make limited queries in each time step
- Must return approximate solution $\tilde{v}_t$
- Goal: Maintain $\tilde{v}_t \approx v_t$
Related Models

- Dynamic Data Structures
  - Similar models of gradual change
  - The algorithm immediately observes the change, has to update a data structure
  - Should be able to answer queries fast with the DS

- Property Testing
  - Solve a problem without reading the entire input
“Sort Me If You Can”, Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian and Eli Upfal, ICALP 2009.

- Want to keep track of a sorted list of objects, whose natural ordering changes over time.
- Can compare a pair of objects at a time.
- Motivated by applications in public opinion polling on websites like bix or youtube slam.
Create, Enter, and Judge

What's the best Trilogy ever made?

The Godfather Trilogy
The Bourne Trilogy

VS

featured faceoff

pick
pick

Browse contests

All Categories
A Cappella
Art
Beauty
Comedy
Photography
Other

Hottest debates

Top 10 Dogs Battle

Golden Retriever
Bulldog

Who is the Sexiest Woman Alive?

Ashwarya Rai
Jessica Alba

All Time Greatest Superhero

Superman
Batman

Popular faceoffs

Qual mulher brasileira é a mais bonita e atrativa?
New to Slam?

This is the place for discovering talented amateur singers, the most adorable clips and the craziest videos on YouTube. Watch pairs of videos and vote for your favorite. Videos are scored based on your votes, and the best videos are featured on the slam leaderboard.

Start voting today and be the first to discover the best videos on YouTube!

Do you have an interesting idea for a Slam? Create one!

Slam Leaderboards

<table>
<thead>
<tr>
<th>Slam</th>
<th>View the winners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bizarre Slam</td>
<td></td>
</tr>
<tr>
<td>Music Slam</td>
<td></td>
</tr>
<tr>
<td>Dance Slam</td>
<td></td>
</tr>
<tr>
<td>Cute Slam</td>
<td></td>
</tr>
<tr>
<td>Comedy Slam</td>
<td></td>
</tr>
</tbody>
</table>
Aggregating the public opinion

- Every time a user visits the site, she is asked to compare two options.
- Need to compute the aggregated “public opinion ranking” over time.
- The public opinion changes over time.
- Non-trivial, even assuming that each user correctly compares the given pair according to the public opinion.
Tracking the public opinion

**Challenges:**

- Public opinion changes over time
- Limited access to public opinion through polling

**Theoretical Problem:**

- Maintain a sorted order of a set of elements
- True ordering changes slowly over time
- Objective: Maintain approximate order subject to bound on comparisons in every time step
Stochastic Permutation Model

- Permutation of $n$ elements evolving over time
- At time $t$, true permutation $\pi_t$
- At every time step a random **consecutive** pair swaps order
- **Goal:** Output a permutation $\tilde{\pi}_t \approx \pi_t$
- Algorithm can query one pair at every step

Kendall-Tau Distance

$$KT(\pi_t, \tilde{\pi}_t) = \# \text{ incorrect pairs } = O(n^2)$$

$$= \# \text{ swaps of consecutive elements}$$

for $\pi_t \rightarrow \tilde{\pi}_t$
We want $\tilde{\pi}_t \approx \pi_t$ to be small.

Kendall-Tau distance:

$$d(\pi_1, \tilde{\pi}_t) = KT(\pi_t, \tilde{\pi}_t) = \# \text{ incorrect pairs} = O(n^2)$$
Results

Sorting
- Lower bound: $\Omega(n)$
- Algorithm giving error: $O(n \ln \ln n)$
  - Based on a simpler algorithm giving error $O(n \ln n)$

Selection
- Algorithm returning element of rank $k + o(1)$
Lower Bound

Theorem
Any algorithm returns a permutation \( \tilde{\pi}_t \) s.t.

\[
E[KT(\pi_t, \tilde{\pi}_t)] = \Omega(n)
\]

Proof idea
- Consider \([t - n/8, t]\)
- We can query \(\leq n/8\) pairs = \(n/4\) elements
- Those are adjacent to \(\leq n/2\) elements
- There are \(n/4\) adjacent elements we know nothing about
- Each swaps with constant probability in \([t - n/8, t]\)
O(n ln n) Algorithm

SimpleAlgorithm:
- Repeatedly run quicksort
- Return latest finished permutation $\tilde{\pi}_t = \tilde{\pi}_{t_1}$

Theorem. SimpleAlgorithm satisfies for all $t$:

$$KT(\pi_t, \tilde{\pi}_t) = O(n \ln n).$$
Analysis

- Easy (wrong) proof: it takes $O(n \ln n)$ steps to sort, in each step at most one pair is swapped, so the distance between the permutations at the beginning and the end of each phase is at most $O(n \ln n)$.

- Wrong: the sorting algorithm needs to work with incorrect, sometimes even inconsistent data. This can create a cascading sequence of errors.

- Quicksort is special!
Quicksort - reminder

- Quicksort(A):
  - Pick a random element $x$ of $A$ as the “pivot”
  - Compare this element against other elements of $A$
  - Recursively sort elements that are less than $x$ and those that are greater than $x$.

- A property of quicksort:
  - *if $a$ is placed before $b$ in the sorted order, either $a$ is compared to $b$, or there’s an $x$ such that $a$ is compared to $x$ and $x$ is compared to $b*. 
Analysis

- Error: \( KT(\tilde{\pi}_t, \pi_t) = KT(\tilde{\pi}_{t_1}, \pi_t) \leq KT(\tilde{\pi}_{t_1}, \pi_{t_1}) + KT(\pi_{t_1}, \pi_t) \leq KT(\tilde{\pi}_{t_1}, \pi_{t_1}) + O(n \ln n) \)

- Study error at \( t_1 \)

- Error = \# pairs where
Analysis

How did we end up with error?

Two cases:

Case 1: $\pi_t$ : 

Case 2: $\pi_t$ : 

True order switched

Not switched
Analysis

**Case 1:** True order switched

- Total steps in \([t_0, t_1]\) = \(O(n \ln n)\)
- One pair swaps per step

Total Case-1 pairs = \(O(n \ln n)\)
Analysis

Case 2: True order never switched

\[ \pi_t : \quad \cdots \quad \cdot \quad \cdots \quad \cdot \quad \cdots \]

There is another (pivot) element that caused the error
Quicksort

![Quicksort diagram]
There is a pivot element \( \bullet \) that caused the error

At some point, in true order:

- \( \bullet \) is pivot and we end up:

  \[ \begin{array}{ccccccccc}
  \circ & \bullet & \circ & \circ & \circ & \circ & \bullet & \circ & \circ \\
  \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
  \end{array} \]

- \( \bullet \) was chosen to swap with each of the two elements \( \circ \), \( \bullet \)

We charge the cost of the pair to the pivot
Analysis – Counting

Quicksort tree

\[
\begin{align*}
\text{E[pivot swaps]} & = cn \cdot \frac{2}{n} = 2c \\
\text{E[pairs]} & = \binom{2c}{2} < (2c)^2 \\
2 \cdot c \frac{n}{2} \cdot \frac{2}{n} & = 2c \\
4 \cdot c \frac{n}{4} \cdot \frac{2}{n} & = 2c \\
\text{# pairs} & = O(\ln n)
\end{align*}
\]
Putting Together

- Case 1: True order has switched – \( O(n \ln n) \)
- Case 2: True order not switched – \( O(\ln n) \)

\[ KT(\pi_{t_1}, \tilde{\pi}_{t_1}) = \# \text{ incorrect pairs} \]

- Case 1: True order has switched – \( O(n \ln n) \)
- Case 2: True order not switched – \( O(\ln n) \)

Total = \( O(n \ln n) \)
O(n ln ln n) Algorithm

- Quicksort runtime = O(n ln n) \quad \text{error} = O(n \ln n)
- No sorting algorithm can sort an arbitrary array with a runtime o(n \ln n).
- However, at the end of Quicksort, each element is only O(ln n) from its correct rank.
- Such “almost sorted” arrays can be sorted faster!
Sorting for the almost-sorted

- Assume each element is within $\ln(n)$ of its correct rank.
- Divide the array into $n/\ln(n)$ blocks of length $\ln(n)$.
- Run Quicksort on each block, and also on blocks shifted by $\ln(n)/2$ positions:

Running time: $\frac{n}{\ln n} \times O(\ln n \cdot \ln \ln n) = O(n \ln \ln n)$

What remains:
1. analyzing this algorithm in the dynamic model
2. Dealing with *accumulating errors*
Dealing with Time

- Ideally we run a global quicksort and then a series of small quicksorts one after another:

- Eventually elements will drift away so we reset with a global quicksort
- But while running it error becomes $O(n \ln n)$
- **Trick:** Execute both independently in parallel
  - **Odd steps:** Regular quicksort
  - **Even steps:** Series of small quicksorts
Parallel Execution

- The output of the algorithm is always the output of the $O(n \ln \ln n)$ sort.
- The output of the $O(n \ln \ln n)$ sort is used as the input to the faster sort.
Sorting – Recap

**Model**
- Real permutation $\pi_t$ swaps a random consecutive pair each time step
- Algorithm can query 1 pair in every step
- Returns a permutation $\tilde{\pi}_t$ close to $\pi_t$
- Kendall tau distance: $KT(\pi_t, \tilde{\pi}_t) = \#$ incorrect pairs $= O(n^2)$

**Results**
- Lower bound: $KT(\pi_t, \tilde{\pi}_t) = \Omega(n)$
- Simple algorithm: $KT(\pi_t, \tilde{\pi}_t) = O(n \ln n)$
- More complicated algorithm: $KT(\pi_t, \tilde{\pi}_t) = O(n \ln \ln n)$
Finding Element at Rank $k$

**Same model**
- Real permutation $\pi_t$ swaps a random pair each time step
- Algorithm can query 1 pair in every step
- **Goal:** Return an element $e$ and minimize $|\text{rank}(e) - k|$

**Results**
- The Sorting algorithm gives a bound of $O(\ln \ln n)$.
- Special case $k = 1$ (finding minimum):
  - Simpler algorithm: compare min with a random element and replace if that element is smaller
  - Defines a Markov chain on the rank of the output. Simple MC analysis shows rank is at most 2 in exp.
Finding Element at Rank $k$

- Algorithm with: $|\text{rank}(e) - k| = o(1)$

- Based on the Motwani-Raghavan median algorithm:
  - $R = n/\ln(n)$ random elements
  - Quicksort($R$).
  - $C$ = elements between $|R|/2 - n^{1/2}$th and $|R|/2 + n^{1/2}$th element of $R$
  - Quicksort($C$). Median is the $L$’th element of $C$, for some $L$.

- This can be adapted to the dynamic setting using the odd-even time steps trick:
  - In odd steps, sort $R$ and compute $C$ and $L$
  - In even steps, continuously sort $C$. 
Algorithms on Evolving Graphs

- **Model:**
  - **Input:** graph $G$ with $n$ vertices and $m$ edges
  - **Change:** in each step,
    - a random edge of $G$ is removed, and
    - an edge is added between a random pair of vertices
  - **Query:** can query the neighborhood of a vertex

- **Problem:**
  - Maintain a path between two given nodes $u$ and $v$, such that the probability that the path is invalid at any point is small.
Algorithms on Evolving Graphs

- It is possible to achieve an error probability of $O(\log n / n)$.
- Almost matching lower bound, within a factor of $(\log \log n)^2$.
- Also, minimum spanning tree and page rank.
Evolving PageRank

- **Change model**: pick a random edge, move its head to a new vertex, chosen with probability proportional to current PR.
- **Probe model**: probe a node, see all outgoing links.
- Want a vector with small $l_1$ dist to true PR.
- Result: can get $O(1/m)$ using Proportional Probing.
Experimental evaluation
Conclusion

- Evolving data sets is an interesting and useful model of computation.

- Open problems:
  - Finding an $O(n)$ algorithm for sorting
    - Conjecture: the randomized algorithm that compares a *random* consecutive pair at each time step achieves this bound.
  - Other problems:
    - clustering/community finding in social networks
  - Imposing continuity constraints on the output
Thanks!