Planar Point Location Using Persistent Search Trees

N. Sarnak & R. Tarjan

CSci 5421: Advanced Algorithms and Data Structures
Ravi Janardan
November 30, 2017

Outline

- Notion of (data structure) persistence
- Motivating application (point location)
- “Sweep + Persistence” paradigm
- Making Red-Black trees persistent
- Amortized analysis
- Discussion

Data structure persistence

- **Ephemeral structure:** Can query (and update) only current version.

![Diagram showing update sequence](image)

- **Persistent structure:** Can query any version (and update current).

![Diagram showing persistent structure](image)

- **Easy persistence:**
  - Save update sequence and rebuild (high query time, low space)
  - Copy entire structure (low query time, high space)
  - Can we get the best of both worlds? [+] low update time] Yes!

Persistence (contd.)

Other notions of “persistence” in CS

- Word processors (undo/redo)
- Version control systems (RCS)
- Programming languages, OS (save state history)
- Non-volatile memory (?)

A new algorithm design paradigm: “Sweep + Persistence”
Planar point location

Given a 2D map (or “planar subdivision”) . . . locate region containing a query point.

Generalizes 1D binary search.

**Real-world application:**

z.umn.edu/obamaromney

Planar point location (contd.)

Proximity queries: Find hospital nearest to my house.

Build Voronoi Diagram (planar subdivision) and point-locate in this.

Planar point location (contd.)

- **Performance metrics** \((n = \text{subdivision “size”, e.g., #edges})\)
  - Storage (desire \(O(n)\))
  - Query time (desire \(O(\log n)\))—many queries
  - Preprocessing time—one-time (desire small)
- **History**
  - 1976: Dobkin & Lipton \((n^2, \log n)\)
  - 1977: Lee & Preparata \((n, \log^2 n)\)
  - 1977: Lipton & Tarjan \((n, \log n)\)—very complicated!
  - 1983: Kirkpatrick \((n, \log n)\)
  - 1984: Edelsbrunner *et al.* \((n, \log n)\)
  - 1986: Cole \((n, \log n)\)—offline setting.
  - 1986: Sarnak & Tarjan \((n, \log n)\)—our paper

Starting point · · · Dobkin-Lipton

- Create strips. For each, store vertical order of segments in Red-Black tree.
- Locate query point via two binary searches \((x\text{ and } y)\); \(O(\log n)\) time. **How?**
- Space bound?
- Pathological subdivision; needs \(\Theta(n^2)\) space.
Key observation

- Successive strips are “similar”, so store only incremental changes. $O(n)$ total changes. Why?
- Sweep over subdivision; create single persistent R-B tree by inserting/deleting segments (“events” at strip boundaries).
- Query: Binary search on $x$ (time-stamps) to find “correct” version. Then binary search in this version with $y$. $O(\log n)$ time.

Elements of “Sweep + Persistence” paradigm

- Treat one axis (say, horizontal axis) as “time”.
- Initialize a Red-Black tree, $T$, to empty and sweep over time-axis with a vertical line.
- At event points, insert/delete appropriate “objects” in $T$ persistently. After the sweep, $T$ will encode succinctly the different versions of the R-B trees generated during the sweep.
- To answer a query, $q$, access the most recent version in $T$ that is no later than the “time” associated with $q$ and query this appropriately (as you would the ephemeral version at that time instant).

Can also choose vertical axis as time-axis. Generalizes to higher dimensions and to other (available) persistent data structures.

How to make an R-B tree persistent?

Wish List:
$O(1)$ space overhead per incremental change $\Rightarrow O(n)$ space overall.
$O(\log n)$ query time—any version.
$O(\log n)$ update time—current version

Three increasingly sophisticated approaches
- Path-copying (log space/change and log query time)
- Fat-node method (constant space and log-squared query)
- Limited node-copying (constant space* and log query)
  * amortized bound per change $\Rightarrow$ overall space is $O(n)$ worst case.

Path-copying method

- Initial tree, at time 0.
- Tree at time 1, after inserting $E$.
  Rule: Copy a node if it points to a node that has itself been copied (or is new) $\Rightarrow$ entire path copied.
  $O(\log n)$ space per update.
  Note: No parent pointers!
- Tree at time 2, after inserting $M$.
- Tree at time 3, after inserting $C$.
- Querying: Locate “correct” root and search “corresponding” tree.
  $O(\log n)$ time. How?
- Updating: Standard way, but on current tree. $O(\log n)$ time.
**Fat-node method**

- Initial tree, at time 0.
- Tree at time 1, after inserting **E**.
  **Rule:** Nodes have unlimited number of pointer fields. Instead of copying nodes, add pointers. $O(1)$ space per update.
- Tree at time 2, after inserting **M**.
- Tree at time 3, after inserting **C**.
- **Querying:** From root follow “correct” time-stamps on links. $O(t \log n)$ time if $t$ different time-stamps.

**Limited node-copying method (hybrid)**

- Initial tree, at time 0.
- Tree at time 1, after inserting **E**.
  **Rule:** Each node has one extra slot for a pointer. If slot is empty, then add pointer; if full, then copy node. 
  Copying can cascade! Space = ?.
- Tree at time 2, after inserting **M**.
- Tree at time 3, after inserting **C**.
- **Querying:** Locate “correct” root. Search “corresponding” tree, following “appropriate” time-stamps on links. $O(\log n)$ time.

**Space bound—Amortized analysis**

In the worst-case, an update can take $O(\log n)$ space.

But . . . maybe worst case does not happen too often?

Yes!

But . . . how to quantify this?

Distribute (or amortize) the total space cost over all updates ⇒ costs average out (expensive updates cost “much less”, cheap updates cost “a bit more”).

**Note:** Still a worst-case analysis, just counting more carefully.

**Amortized space bound—Intuitive analysis**

Effect of update $u_i$:
- May create a new non-full node and causes $k \geq 0$ full nodes to get copied ⇒ $k + 1$ new, non-full nodes ($k$ can be large!)
  - So, $k + 1$ units of space used.
- Causes $O(1)$ non-full nodes to get full. Because only $O(1)$ rotations!
  - Associate this event with $u_i$.
- Thus, every full node is associated with an update, and every update with $O(1)$ full nodes.
- So, can “charge” space for each of the $k$ nodes copied during $u_i$ to a previous update $u_j$ ($j < i$), and each such update is charged at most $O(1)$ times.
- At most $O(1)$ units of space charged to each of $O(n)$ updates (including newly created node), so total space used is $O(n)$ ($n =$ subdivision size).
- **Note:** Average (or amortized) space cost per update is $O(1)$ units, even though a given update can cost a lot more ($k$ units).
Intuition for $\Phi$?
Define $\Phi$ as number of full nodes in current tree.
(Non-negative!)

Update causes:
- a new non-full node to possibly be created. No change in $\Phi$.
- $k \geq 0$ full nodes in current tree to get copied to non-full nodes
  $\Rightarrow$ $\Phi$ decreases by $k$ (since these full nodes are no longer in
  current tree and new nodes in current tree are non-full).
- $O(1)$ non-full nodes in current tree to get full $\Rightarrow$ $\Phi$ increases
  by $O(1)$.

So, $\Delta \Phi = -k + O(1)$.

Actual space cost is $c_i = k + 1$.

Amortized space cost is $\hat{c}_i = c_i + \Delta \Phi = O(1)$

Contributions/Strengths:
- Integrates three key concepts:
  - Persistence (data structuring technique, first for RB-trees)
  - Sweep (algorithm design paradigm)
  - Amortization (analysis method)
- Solves a fundamental problem (planar point location).
- Approach has broad applicability (hive-graph killer, 3D point
  loc., generalized intersections)
- Exposition (clear(?), balanced)

Weaknesses:
- How original?
- How practical?

How did they do it?
Hard to tell ... combination of deep domain knowledge,
experience, intuition, focus, patience, and ... persistence(!)

Another example

Given $n$ horizontal line segments in the plane ... output segments
interacted by a vertical query line.

Demo applet: z.umn.edu/persistentdemo