CS 261: Data Structures

Week 9: Persistence

Lecture 9b: Fat nodes and persistent search trees

David Eppstein
University of California, Irvine

Spring Quarter, 2023

Review of persistence

Persistence

Classical data structures

Handle a sequence of update and query operations

Each update changes the data structure

Once changed, old information may no longer be accessible

Persistent data structures

Each update creates a new version of the data structure

All old versions can be queried and may also be updated

Driscoll, Sarnak, Sleator, and Tarjan,
"Making data structures persistent", STOC 1986

Types of persistence

Partial persistence

Updates operate only on latest version of structure

Queries can examine old versions

History is linear (sequence of operations forms a single timeline)

Full persistence

Updates can be applied to any version

History forms a tree
(updating an old version creates a new branch)

Confluent persistence

Updates can combine multiple versions (like git merge)

History forms a directed acyclic graph

Path copying

Requires a tree-like data structure in which each node can be reached by a unique path from a root node

Represent each version as a pointer to its root node

Query: same as non-persistent, starting from the version's root

Update: make new copies of all nodes on paths from root to changed nodes, with links to old unchanged nodes

Path-copying analysis

Query and update time: Same as non-persistent

Space: Same as total update time

May be significantly bigger than non-persistent space

Today's goal: implement persistence differently

Work for any structure, not just tree-like structures

Space = O(changed data in update), not O(total update time)

Fat nodes

What are fat nodes?

General technique for making any data structure persistent

- Divide the structure into pieces with a constant number of words (nodes of a node-pointer structure, cells of an array, individual words of memory)
- Each piece stores the history of what has been stored there
- To access a version of the data structure, simulate a non-persistent operation, replacing each read or write of a piece of data by a query or update to its local history

Typically slower than path-copying because each access to a piece of memory turns into a more complicated data structure operation

More general (doesn't require nodes linked into paths from roots)

Optimal space (only enough to store all of the changes to the non-persistent structure)

Partially persistent fat nodes

In a partially persistent data structure, there is only one sequence of update operations

We can represent a version by a number, its position in the sequence

Each piece of memory stores a collection of key-value pairs

- Key = the number of an update operation
- Value = the value of that piece after that update

In an operation that wants to read or write version i of piece x, we need to find the predecessor of i in the pairs stored for x, or add a new pair (i, x)

Partially persistent fat node analysis

Space = total number of times a non-persistent data structure would write a piece of memory

(May be significantly smaller than total update time if most of the work in an update is reading not writing)

Time per operation =
(non-persistent time) $\times$ (time per predecessor operation)

- If local version-value pairs are stored in a binary search tree, then the time to access a piece of memory in a data structure with n updates is slowed down by $O(\log n)$ compared to non-persistent structure
- If they are stored in a flat tree (in a version of flat trees extended for predecessor queries) then the time to access a piece of memory is $O(\log \log n)$

Fully persistent fat nodes

In a fully persistent data structure, the history forms a tree

- Tree nodes = versions
- Parent of a version = the version it was updated from

Each update adds a new leaf to the tree

Each piece of memory stores a subset of versions (set of tree nodes); reading the piece of memory requires finding the nearest ancestor in this subset

The details of this nearest ancestor problem involve both maintaining order in lists and flat trees, but can be done with the same $O(\log \log n)$ slowdown as partial persistence.

Implementation

Because it does not depend on the details of the structure, fat-node persistence can be implemented generically, as a wrapper:

Implement your data structure normally, in a non-persistent way

Call a wrapper function to make a persistent version of it

Several Python implementation projects exist

But be careful: "persistence" can also mean that your data survives on disk from one execution of a program to another

Persistent binary search trees

Comparison of persistence techniques

For a partially persistent binary search tree after n updates:

- Path copying uses $O(\log n)$ time per operation but takes a total of $O(n \log n)$ space
- Fat nodes use $O(\log n \log \log n)$ time per operation and are complicated (flat trees) but use only $O(n)$ space
- Hybrid structure (detailed on the following slides) combining both path copying and fat nodes has $O(\log n)$ time per operation, $O(n)$ space, the best of both worlds. And it's much simpler than fat nodes because it doesn't need flat trees.

Hybrid persistent search trees

Versions are numbered (as in the fat node technique)

Pieces of memory = nodes in a WAVL tree

- This is a kind of search tree where each node has an integer rank; constraints on ranks of neighboring nodes $\Rightarrow$ balanced
- We store the node ranks non-persistently, because we only need them to update the latest version of the structure
- Each update causes $O(1)$ rotations changing the tree structure

Each node stores its local structure (left and right children) for up to three versions

When we want to update the structure of a node and its local history is full (already stores three versions), we make a copy of the node and add a new local version at its parent pointing to the copy

Hybrid tree analysis

Because this is only partially persistent, we can use amortized analysis

Potential function Φ = sum over nodes of most recent version of the tree of how many versions are stored at each node

Making a local change in structure and adding a new version increases Φ by one

Making a new node to replace a full node decreases Φ by two (at that node) and increases it by one (at the parent node)

So decrease in Φ cancels extra space used to create new node and the amortized space (not amortized time) per update is $O(1)$

Time per operation = non-persistent WAVL tree operation $\times$ time to find correct version at each node = $O(\log n) \times O(1) = O(\log n)$

The locus method

Method for building data structures for problems where:

Data does not change

Queries are points in the plane

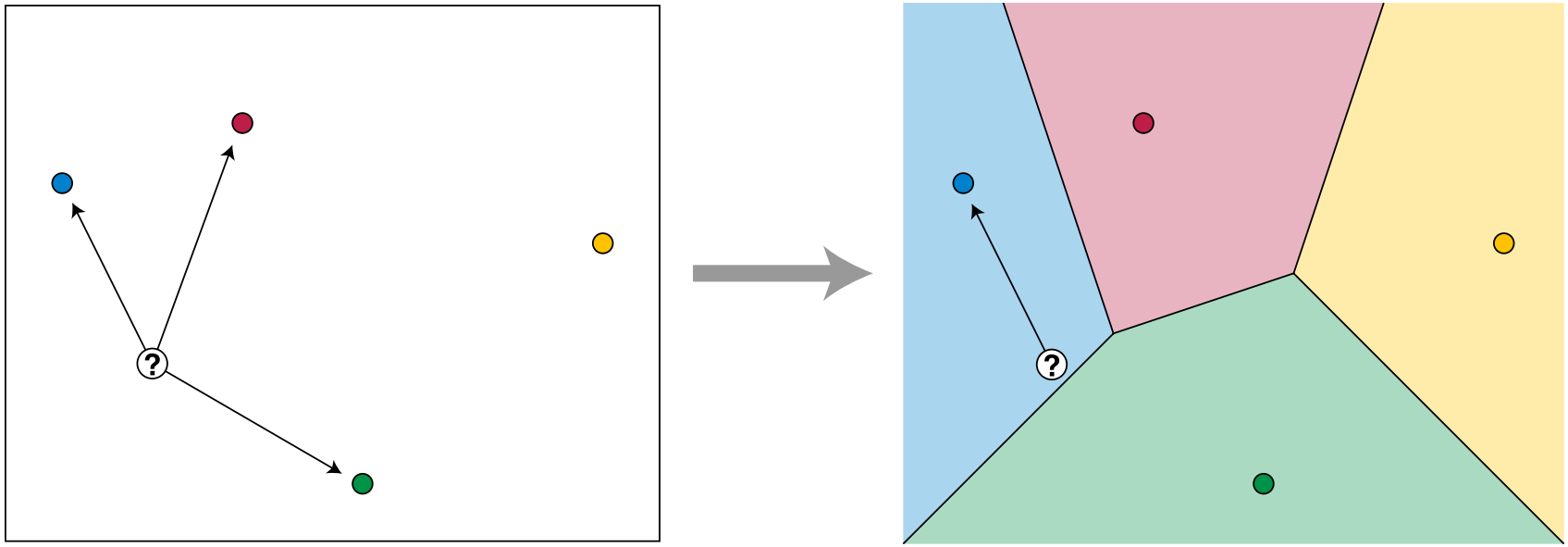
Answers are constant over regions of the plane
(rather than varying continuously)

Partition plane into regions within which answer is constant

Build "point location" data structure
that can find the region containing each query

Post offices and Voronoi diagrams

The post office problem: given a set S of points in plane answer queries asking: which point in S is closest to query point q?



Voronoi diagram: partition plane into regions surrounding each point of S, within which that point is closer than any of the others

Point location in Voronoi diagram = answer to post office problem

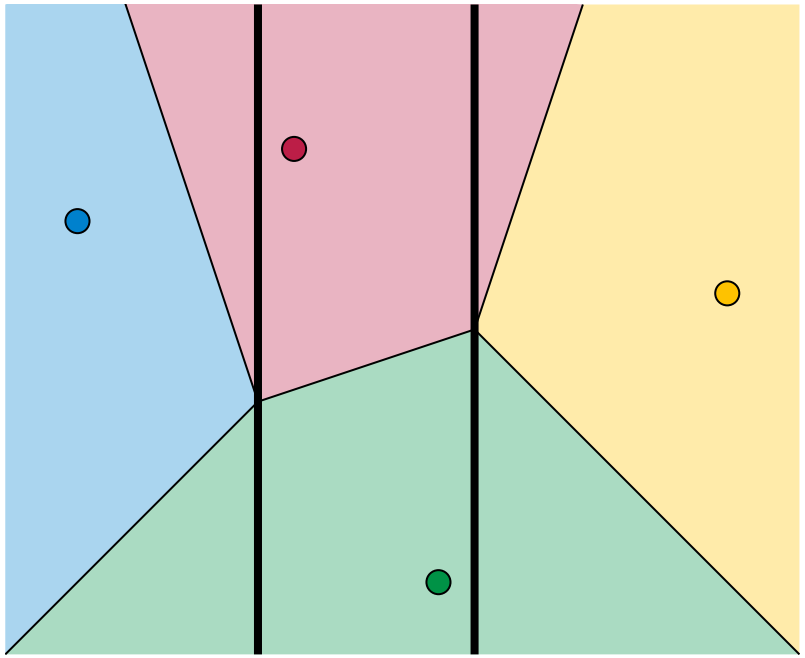
Point location by slabs

Simplifying assumptions: regions are polygons, at most three meet at any vertex, no vertical boundaries

Partition plane by vertical lines through vertices

Point location: binary search among x-coordinates of vertical lines, then binary search in vertical ordering of regions in slab between two vertical lines

Query $O(\log n)$, space $O(n^2)$



Point location by persistent search trees

Vertical ordering of regions changes only by removing a region (blue from left to middle slabs) or inserting a region (yellow from middle to right)

Build partially persistent binary search tree of vertically ordered regions, with one version/slab

Point location: binary search in x-coordinates of vertical lines to find slab and its version, then search in that version of the persistent search tree

Query $O(\log n)$, space $O(n)$