SF2656/FSF3565 Assignment 2

Björn Wehlin

Spatial search

Many scientific and engineering applications rely on spatial queries on point sets. For example, in a CAD application, one may need to be able to select all vertices of a set of shapes that fall within some given region. Likewise, in, e.g., robotics, information about the world may be presented as a 3d point cloud, from which we may be interested in certain geometrical subsets.

Because the point sets may be very large, efficient data structures are important. In this assignment, you will implement a *bucket quadtree* that supports querying which points of a 2d point cloud that lie within an axis-aligned rectangle.

Bucket quadtrees

A *quadtree* is a tree where each parent has four children: northwest (NW), northeast (NE), southwest (SW), and southeast (SE), called *quadrants*. Each quadrant can contain a node, or be empty.

A node whose children are all empty is called a *leaf node* (or just leaf), and the node at the top of the tree (i.e., the one that has no parents) is called the *root node* (or just root).

In a *bucket quadtree*, we define an upper limit for how many points are allowed to reside in a leaf node. Non-leaf nodes (*internal nodes*) do not contain any points.

Construction of a bucket quadtree proceeds recursively. We begin by creating a root node and let R be the minimal axis-aligned bounding box (AABB) over all the points that will reside in the tree (i.e., the smallest rectangle that contains all points in the input).

If the number of points is smaller than or equal to the bucket capacity, we are done; simply insert all points in the root node. If this is not the case, then we subdivide the root AABB, R, into four quadrants. This can be done either by splitting the root AABB in the middle both in x and y, or in some other way, e.g., by considering the median x and y coordinates of the input points.

For each of the quadrants, we repeat the procedure. We take as input the points that fall within the quadrant, and we use the quadrant's AABB as the new "root" AABB.

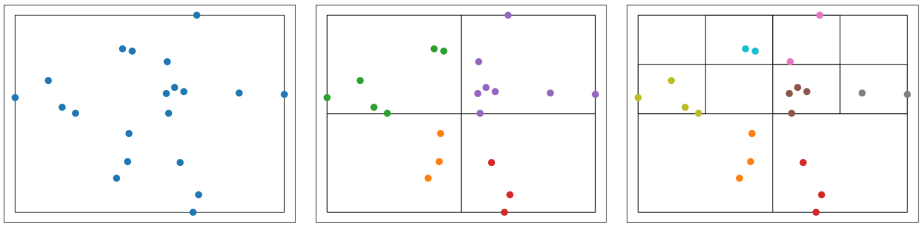

Figure 1: Bucket quadtree construction over 30 random points, with bucket capacity 5. In the first step (left panel), the root node contains all points. Since there are more than 5 points in the root, we subdivide (using the midpoint scheme) into four quadrants. Now (middle panel), the SW and SE children both contain 3 points each, so no further subdivision is needed for these. However, NW and NE both contain more than 5 points, so we subdivide NW and NE (right panel).

Figure 1 illustrates the algorithm for 30 random points and a bucket capacity of 5 points.

For a more interesting example of a quadtree, see Figure 2.

Utitlities

As part of this assignment, we supply a few helper classes and functions, as well as some test data.

- `random_points.hpp`: contains `sf::RandomPointGenerator`, which can be used to generate synthetic test data
- `point_reader.hpp`: contains the function `sf::readCsvPoints` that can be used to read the test point files
- `mpl_writer.hpp`: contains `sf::MplWriter` that is used to create matplotlib-based plotting scripts for Python
- `timer.hpp`: contains `sf::Timer` that can be used to time sections of your code
- `swe.csv`: centroid points of 2,523 Swedish districts (data from Lantmäteriet open data, CC0 – public domain)
- `swelakes.csv`: 299,865 outline points of lakes of Sweden (data from Lantmäteriet open data, CC0 – public domain)

Each `.hpp` file contains documentation of its contained entities, with usage instructions.

For the map data, you may plot a basemap using `contextily`. Simply add the following code on immediately before `plt.show()` in the Python script:

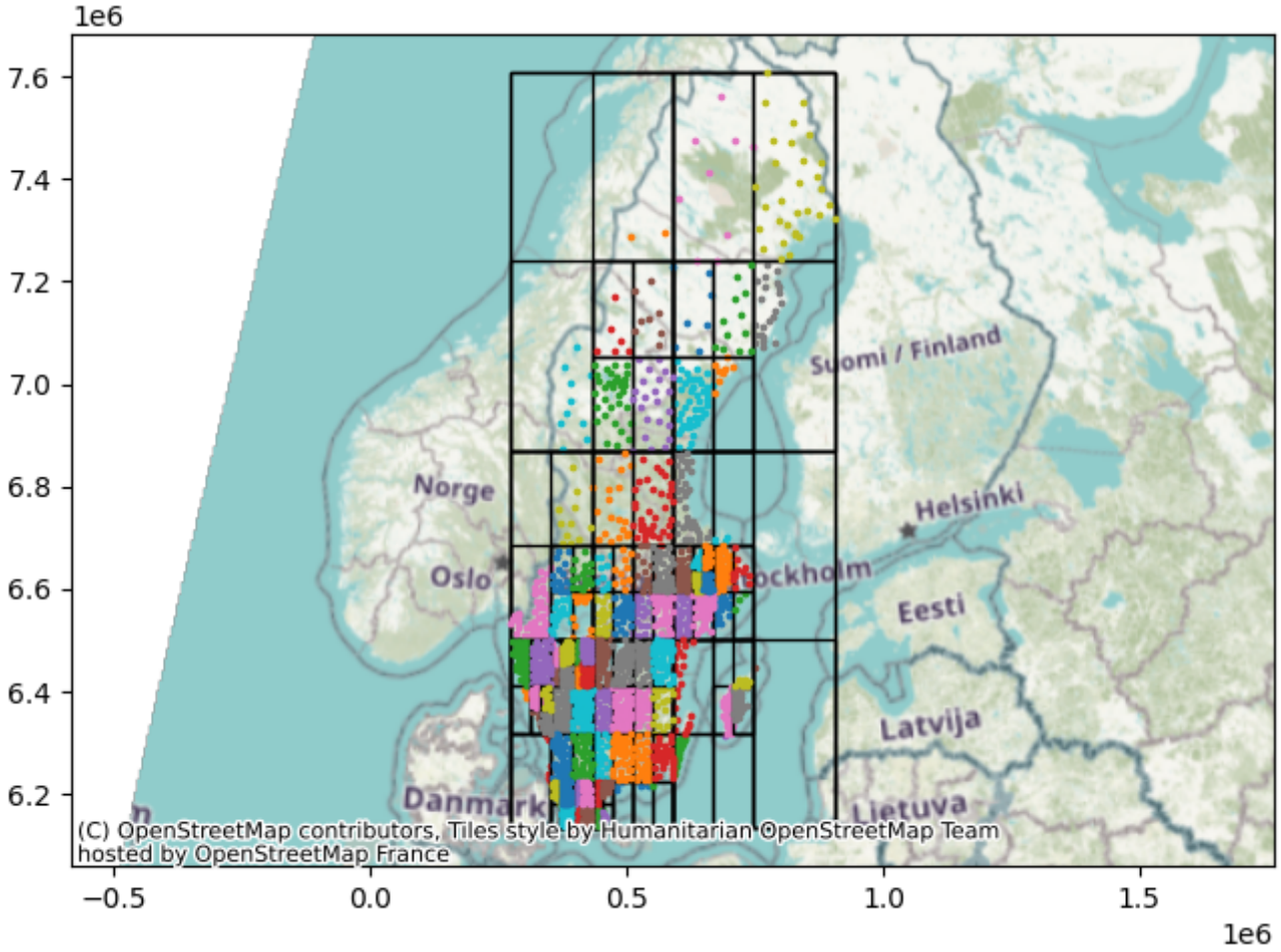

Figure 2: Bucket (capacity 64) quadtree of 2,523 Swedish districts (each point is at the centroid of its district). Data from Lantmäteriet open data (CC0: public domain). Basemap Copyright OpenStreetMap contributors.

```
import contextily as cx
cx.add_basemap(ax, zoom=4, crs='EPSG:3006')
```

A higher zoom level gives more resolution but can be quite slow to load.

Task 1: Geometry classes (0.5pts)

Your first task is to implement two classes: `Point` and `Rectangle`. To work with the supplied helper functions and classes, `Point` should have two public member variables `x` and `y`, and it should be possible both to default-construct `Point`, as well as using `Point(xCoord, yCoord)`.

The `Rectangle` class should have two public member variables, `bottomLeft` and `topRight`—both of type `Point`.

Beyond this, you are free to add anything to these classes. If you wish, you can make them class templates but this is not necessary (this is perhaps the more professional approach, but keep in mind you will then have to design your quadtree class, etc., as a class template, as well).

Task 2: Quadtree implementation (6 pts)

Implement a `QuadTree` class that can be constructed from a collection of `Point`s (e.g., a `std::vector<Point>`). The bucket capacity should be configurable.

Use the supplied `sf::MplWriter` class to create a Python plotting script from your rectangles and points. **Make sure to print all points belonging to a single leaf node as a single vector!** This way, Matplotlib will color the points by their corresponding leaf node.

Task 3: Ownership (0.5 pts)

Recall that ownership refers to who/what is responsible for cleaning up a resource. Explain the ownership semantics in your implementation. Who/what owns the points in the quadtree? Who/what owns the nodes? Is the ownership shared or exclusive?

Task 4: Region query (2 pts)

Add a `query` method to your `QuadTree` class that takes a `Rectangle` as input. The method should return all points that lie inside the supplied `Rectangle` as output.

To obtain points for this task, the `query` method should use the structure of the quadtree, recursively working its way down the tree and stopping when possible.

Plot some representative regions. To do this, you can call

`MplWriter::operator<<(const std::vector<PointT>&)` twice: once for all points in the dataset, followed by one time with the result of your query.

Task 5: Performance (1pts)

Measure the peformance of your region query by generating random point sets of various sizes and querying small regions (so that around $\leq 1\%$ of the total input size is returned).

Also measure and report the time required to construct the quadtree.

How does your query performance compare against simply iterating through the input point vector and checking each point? You may have to use quite large inputs (millions of points).

How does the bucket capacity impact performance?

Do you have any recommendations on when to use your quadtree with respect to input sizes, etc.?

Submission

The programming exercises should be done individually, or in groups of two. Hand in a report containing:

- Comments and explanations that you think are necessary for understanding your program. (But do not comment excessively.)
- The output of your program according to the tasks. Don't forget to draw conclusions!
- Printout of the source code in PDF format. (We need this to be able to comment on your code, points will be deducted for missing to do this.) You can either put the source code as an appendix to your written report, or you can hand in separate PDF(s).

In addition, all source code for your program(s) in .zip/.tar format and instructions for how to compile/run.

Important! Your submission must contain at least the following two files:

- PDF report (not zipped!) with your findings and source code
- .zip/.tar archive containing your source code

Submissions not following this format will not be accepted and will have to be resubmitted (these will be considered late submissions).

Good luck!