Spatial Databases - Course Project

Trapezoidal map for point location

Giacomo Callegari

giacomo.callegari@studenti.unitn.it

207468

January 2020

1 Introduction

Point location is a very common problem in geometry: it consists in finding which face of a planar subdivision S contains a point q of given coordinates. An efficient solution is provided by the trapezoidal map algorithm, which builds a refinement of the subdivision that partitions it into trapezoid-like faces.

This algorithm has been implemented in Python with an object-oriented approach, modeling the relevant classes that describe the geometric elements and the required data structures.

2 Problem statement

A planar subdivision S is a set of non-intersecting segments that may however share endpoints. Segments are assumed to be in general position, which means that no distinct endpoints have the same x coordinate. Another simplifying assumption is that the subdivision is enclosed in a rectangular bounding box R, so that all faces are bounded.

Then, a trapezoidal map T can be built from S by drawing, for each segment endpoint, upper and lower vertical extensions that stop at the first encountered segment: this refinement of the subdivision creates faces that are trapezoids, or triangles in the degenerate case. The search structure D is a DAG whose leaves represent the trapezoids of T, while internally x-nodes reference segment endpoints and y-nodes are associated with segments.

3 Algorithm

3.1 Class diagram

The Python classes are organized as shown in the UML diagram of Figure 1 and have been grouped by domain in the following Python modules:

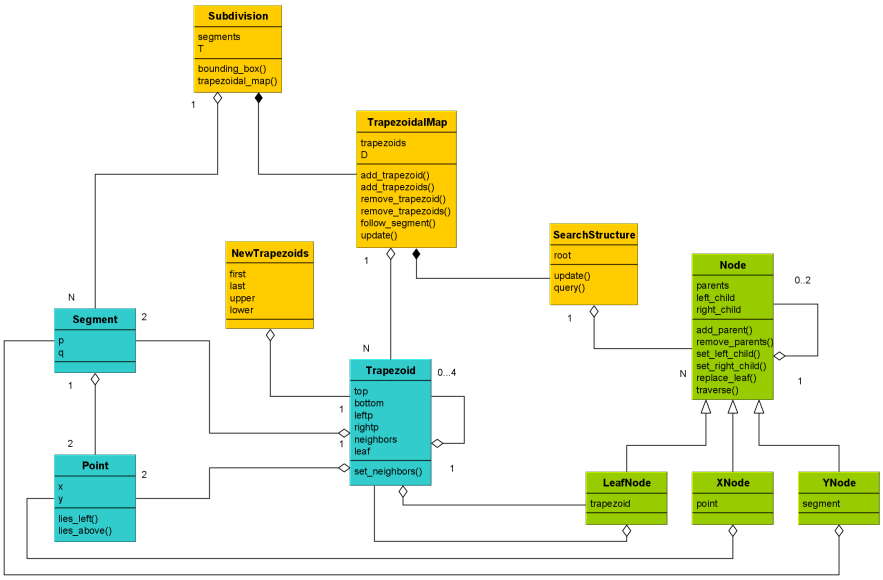

Figure 1: The UML class diagram.

- `geometry` (`Point`, `Segment`, `Trapezoid`)
- `nodes` (`Node`, `XNode`, `YNode`, `LeafNode`)
- `structures` (`Subdivision`, `TrapezoidalMap`,`SearchStructure`, `NewTrapezoids`)

There are also two additional modules:

- `util` contains various utility functions
- `main` represents an example of execution

The classes `Point`, `Segment` and `Trapezoid` define the geometric elements in a 2-dimensional space. Each point has two coordinates and each segment has a left and a right endpoint. Trapezoids are uniquely identified by two segments (`top`, `bottom`) and two endpoints (`leftp`, `rightp`), but can also reference at most four adjacent trapezoids. Each trapezoid also maintains a one-to-one mapping with a leaf node that will be used in the search structure.

The data structures `Subdivision`, `TrapezoidalMap` and `SearchStructure` are required to build the trapezoidal map and perform queries on it, while `NewTrapezoids` acts like a simple container for updated trapezoids. The search structure is supported by a `Node` class that specializes in `XNode`, `YNode` and `LeafNode`: each subclass is treated differently when traversing the DAG because it is associated with a different geometric object.

3.2 Initialization

When creating a `Subdivision` object from a set of segments, the constructor first calls the method `bounding_box()` to obtain the bounding box R. A new `TrapezoidalMap` object T is also initialized in the process: its constructor creates in turn the related `SearchStructure` D.

3.2.1 Bounding box

In the `bounding_box()` method, the extreme coordinates of the subdivision are computed by iterating over all segments, then an arbitrary margin is added to these coordinates to obtain the vertices of the rectangle. The bounding box is finally added to T as the initial trapezoid, with a corresponding leaf in D that is initially the only node.

3.3 Building the structures

The trapezoidal map T and the search structure D are built simultaneously in the method `trapezoidal_map()`, with a process that is incremental and randomized. In fact, in each iteration a random segment of S is added and the intersected trapezoids in T are found. Then, both T and D must be updated by removing the intersected trapezoids and adding the new trapezoids that have been generated by the segment and its vertical extensions.

3.3.1 Adding a segment

The search for trapezoids $\Delta_0, ..., \Delta_k$ that are intersected by the new segment s_i is performed by the method `follow_segments()` of `TrapezoidalMap`. After Δ_0 has been identified by querying the segment's left endpoint on the current version of D, the other trapezoids are found by following right neighbors until the end of the segment. The upper or lower neighbor is selected each time depending from the position of `rightp` with respect to s_i.

The method returns a list of `Trapezoid` objects, ordered from left to right, that will be used to update the trapezoidal map and the search structure. The case of a single intersected trapezoid is handled separately because of its simplicity.

It's possible that the left endpoint p_i coincides with an existing endpoint: in this case, the traversal of the search structure will stop at the corresponding x- or y-node. The query point is then moved to the right by an ϵ along the segment s_i and the traversal is resumed from the last node.

3.3.2 Single intersected trapezoid

When updating the trapezoidal map, in the general case the intersected trapezoid is partitioned into four smaller trapezoids:

- A to the left of p_i

- B to the right of q_i
- C above s_i
- D below s_i

Such trapezoids are inserted in a `NewTrapezoids` object that is required to update the search structure, along with the list of intersected trapezoids. Again, the process is very simple: the old trapezoid's leaf is replaced with a subtree that generally has four leaves, two x-nodes and one y-node.

If the left and/or right vertices of s_i coincide with existing endpoints, A and B will respectively collapse along with the relevant x-nodes and leaves, while the presence of C above and D below s_i is guaranteed.

3.3.3 Multiple intersected trapezoids

In this case, two new trapezoids `first` and `last` are identified to the left and to the right of the segment: again, they may collapse in presence of coinciding endpoints. The m intermediate trapezoids that have been intersected are horizontally split through the function `util.split_trapezoids()`. This method returns a list for the upper parts and another for the lower parts, both of size m; such trapezoids maintain the relevant neighbors of the original elements.

Some of these (upper/lower) parts may now be merged into the same trapezoid if the new segment has shortened some existing vertical extensions. This operation is performed by `util.merge_trapezoids()`, which iteratively checks if two or more adjacent parts share both the *top* and *bottom* segments. The result is a list of size m where the i-th element represents the merged trapezoid that contains the i-th original part: for this reason, there may be consecutive duplicates.

After creating the (possibly merged) trapezoids above and below s_i, their neighbors must be initialized through `util.update_neighbors()`. Such function trivially sets the lower neighbors of trapezoids above s_i and vice versa, but also considers "external" neighbors that are obtained from the split trapezoids. When a trapezoid Δ sets a neighbor Δ', such trapezoid symmetrically sets Δ as neighbor in order to maintain bidirectional references.

Again, a `NewTrapezoids` object is created and passed to the method that updates the search structure. The leaves of the first and the last intersected trapezoids are replaced with two x-nodes in the general case, while every old intermediate trapezoid is replaced by a y-node. The mapping from original to merged trapezoids is obtained from the previous m-sized lists and used to set the children of the y-nodes.

3.4 Querying a point

Once all segments have been added and D is complete, such structure can be used to find the trapezoid that contains a query point q. The search starts from the root of the DAG, goes through several inner nodes and finally ends in the leaf that corresponds to the desired trapezoid.

If an x-node is encountered, it represents a segment endpoint: the query selects the left child if q is above the endpoint, the right child otherwise. A y-node represents a segment instead: the left child is chosen if q is above the segment, the right child if it is below.

The traversal may stop at a x- or y-node if q is an existing endpoint or lies on a segment. In these cases, the query fails as it cannot determine the corresponding face.

4 Conclusions

The trapezoidal map algorithm requires $O(n \log n)$ time to build the data structures that describe a planar subdivision of n segments, while the query time is $O(\log n)$. The space complexity is improved w.r.t. the slab algorithm, as the cost has gone from quadratic to linear: there are in fact at most $6n + 4$ vertices and $3n + 1$ trapezoids in the trapezoidal map.

It should be noted that the trapezoidal map is obtained deterministically from S, while the search structure is affected by the order in which new segments are added. However, this is not a problem, since at every iteration T and D are valid for the current set of segments.

Regarding the implementation, the most complex part was the initialization of the neighbors of new trapezoids. In particular, the leftmost and rightmost new trapezoids had to be handled separately from the intermediate ones, with many possible edge cases to consider. Such evaluation was also necessary in the update of the search structure, but to a lesser extent.

The Python code of the implementation can be found on GitHub[1], along with this report, the UML diagram and two images representing the examples in `main.py`.

[1] `https://github.com/giacomocallegari/spatial-databases-project`