Point Location in Voronoi Diagrams

Anders Høst Kjærgaard and Hildur Uffe Flemberg
{ahkj, hufl}@itu.dk

IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

Abstract. Voronoi diagrams are used in a variety of contexts in mathematics, biology and computing. We are interested in analysing the composition of the voronoi diagram to say something about the underlying data that the diagram represents. In this paper we descibe a solution that generates a voronoi diagram from an arbitrary set of data points and returns a data structure for performing point location i O(log n) time in the generated diagram. The solution makes use of an existing open-source implementation of Fortunes algorithm for generating voronoi diagrams, which is merged with our own implementation of a trapezoidal map for performing point location. Querying the trapezoidal map data-structure will return the site of the voronoi cell in which the query point lies. Secondly, we use the point location as basis for an analysis of the expected number of query points that will end in the biggest voronoi cell when distributed randomly across the diagram. We discuss why performing point location on Voronoi diagrams could of interest in various domains.

Keywords: algorithms, computational geometry, point location, software development

1 Introduction

Voronoi diagrams have been know for... In YYYY Fortune invented the algorithm for ... They are interesting because this and that. Get examples from `http://voronoi.com/wiki/index.php?title=Voronoi_Applications` We use restaurants, defined as sites.

Various interesting questions could be asked about the diagram, that would also pose of interest in algorithm design. In which cell does a given point reside? Which cell is the largest? The questions are not jsut interest for the algorithmic challange, but also due to how we can interpret the answer to such questions in relation to the sites that composit the diagram. E.g. we could assume that if a given

2 Background

Basic description of voronoi diagrams and the algortihm. Refer to the book, but explain running time of Fortune vs slap approach and single shot. Describe circle events, how they end up and get pulled from the event queue and why they are

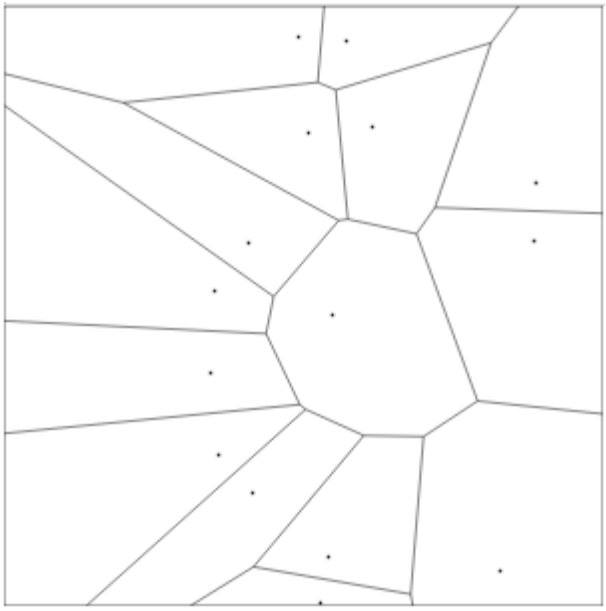

Fig. 1. A Voronoi Diagram with 15 sites

important - segments are generated and we want to add them to the map when taht happens.

Trapezoidal map. Maybe a bit more fine grained explanation. Refer to book for full detail, a few visual examples.

3 Problem Statement

As mentioned in section, there are multiple useful applications of point location in voronoi diagrams. In the remainder of this report we describe a solution that facilitates this kind of point location using techniques from two algorithms that are known in advance – one that generates voronoi diagrams and one that performs point location in any planar subdivision.

3.1 Desirable Features

- The complexity should be unchanged, i.e. not bigger than any of the existing algorithms
- The solution must be generic, i.e. not targeted a specific business area or domain

3.2 Assumptions

- We require that no two sites of the input share the same x-coordinate as this is a simplifying assumption made by the original point location algorithm.

This assumption means that every trapezoid can be uniquely represented in terms of their left point, right point, bottom segment and top segment. Furthermore, every trapezoid can have at most four neighbours. It is clear that this assumption is unreasonable when working with real-world data. How to deal with input where such "degenerate" cases occur is described in [1]

4 Solution

The provided solution consumes a set of 2D points P and returns D for performing point location queries in a Voronoi diagram corresponding to P. The attached CD contains the implementation as C# source code that builds against .NET 4.0. An example providing P and querying D can be found in the program main. A parser and example for generating data points from the geographical coordinates of a set of fast-food restaurants is included.

The solution is based on the description provided in [1] on point location and Voronoi diagrams, and as such, we do not cover the everything in detail, but rather provide highligts of parts that we found to be tricky to get right. The implementation of Fortunes algorithm is an open-source implementation that can be found at [cite to misc and codeplex project] . Also , the solution includes a simple GUI from [cise to misc and ref to other os project] that can be useed to visualize simple small-scale datasets. The implementation of T and D is provided by the authors and our main work constitues in implementing these, besides briding with Fortunes algorithm.

One way of accomplishing what we want would be to simply take to the output of Fortunes algorithm, that is, the generated edges, and give them as input to the algorithm for the trapezoidal map. Both algorithms run in O(nlogn) so the the complexity of running the two in sequence is the same. Alternatively we could add the edges to T on the fly as the are found by Fortunes. It is easy to see that we will get the same complexity as we are virtually doing the same thing. To show this easy connecting of the two algorithms, the implementation demonstrates the latter.

4.1 Point location in Voronoi

D is implemented with a standard composite-pattern, and offers a $Find(point)$ operation that will return a bounding trapezoid for the query point. Recall that a trapezoid is simply defined by its side points and top and bottom segment. And as each segment is just a segment w egot from the Voronoi diagram, it holds a reference to the site of the voronoi cell. Fortunes algorithm adds this edge to site reference, and, eventually, it is the only way that trapezoid can tell which Voronoi cells it lies in. Of course, queies will only be precise when all segments have been added, and all trapezoids are completely trapped inside Voronoi cells.

Fig. 2. Inserting a segment in T that is contained inside a single existing trapezoids

Each time a segment is discovered for the Voronoi diagram, it is inserted into the trapezoidal map, and T is updated with new subtrees according to the new trapezoids that appear from the inserting the new segment. A this point a lot or re-wiring occurs, as new and existing trapezoids need to have their neighbourhoods re-wired correctly. One case of this is as seen in Figure 2.

When the segment is contained in an existing trapezoid where we just need to make sure that we indetify what new trapezoids are created from inserting the segment, and updating D with a corresponding subtree, which is somewat trival. In the case where a new segment crosses several existing trapezoids, things get a bit more tricky, and we will in the following explain how it can be solved.. The situation is depicted in Figure 3.

[1] provides an good high-level description of how to handle this situation and states that it is done in linear time, which is possible as we can find neighbouring trapezoids by jumping from one trapezoid to the next by neighbour references - we do not need to look up neighbouring trapezoids in D. Now to handle the different case of the position of the inserted segment, we can start out by taking care of the boundries when one or both of the new segment's endpoints are connected to existing vertices, which is handled much like the the contained case. Secondly, we have to create new trapezoids that are, in some sense, horizontial merges of existing trapezoids. As an example, in Figure 3, we create trapezoid C with a left point from $\Delta 1$, a right point that will be the right most vertex of the newly inserted segment left most vertex, top being the new segment, and

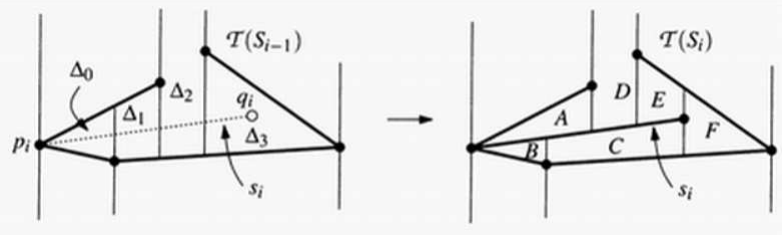

Fig. 3. Inserting a segment in T that crosses several trapezoids

bottom being the the segment shared by $\Delta 1$, $\Delta 2$ and $\Delta 3$. Our solution to this is to always handle $\Delta 0$ and $\Delta 3$ in isolation, that is, add trapezoids much like with the case of the contained segment, and set the neighbourhoods. Secondly we do a recursive merge of the new trapezoids above the inserted segment, and under the segment. The pseudo code for doing an upper merge is show in Algorithm 1.

Doing a merge of the trapezoids below the inserted segment is done the same way, but with the neighbouring reference flipped horizontially. The last thing we need to do is to update D.

4.2 Expected Customers

5 Evaluation

Test with a few screen shots showing that algo actually found the right cell.

6 Threats to Validity

Buggy software til generering af Voronoi.

Algorithm 1: MergeUpper

input : A trapezoid *start*, a Segment *segment*, and a point $leftPoint$
output: A list of trapezoids
Define LLN, ULN, LRN, URN as Lower Left Neighbour, Upper Left Neighbour, Lower Right Neighbour, Upper Right Neighbour of a trapezoid respectivly;
begin
    Create new trapezoid t;
    $t.leftPoint \leftarrow leftPoint$;
    $t.top \leftarrow start.top$;
    $t.bottom \leftarrow$ segment;
    $cur \leftarrow start$;
1   **while** *cur.rightPoint is below segment and cur is on the segment* **do**
        $cur \leftarrow$ cur.URN;
2   **if** $segment.RightMostVertex.X < cur.RightPoint.X$ **then**
        $t.RightPoint \leftarrow$ segment.RightMostVertex;
        else $t.RightPoint \leftarrow cur.RightPoint$;
3   **if** *cur.rightPoint is before end of segment* **then**
        next $\leftarrow MergeUpper(cur.LRN, segment, cur.rightPoint)$;
        $t.LRN \leftarrow next.First$;
        $t.URN \leftarrow cur.URN$;
    return t *and next*;

Algorithm 2: InsertMergedTrapezoids

input : A list of existing trapezoids *olds*, a list of upper merged trapezoids *upper*, and a list of lower merged trapezoids *lower*, and the inserted *segment*
Result: Updated search tree D
begin
    let $i \leftarrow j \leftarrow k \leftarrow 0$;
1   **while** $k < olds.Count$ **do**
2   **while** $upper[j].RightPoint.X < olds[k].RightPoint.X$ **do**
            $j \leftarrow j + 1$;
3   **while** $lower[i].RightPoint.X < olds[k].RightPoint.X$ **do**
            $i \leftarrow i + 1$;
        Replace $olds[k]$ with subtree of *segment*, $upper[j]$ and $lower[j]$ in D
        $k \leftarrow k + 1$;

7 Plan For Future Work

7.1 Future Work

Create a clean and generic voronoi implementation in .NET 4.x Find best point for planar location. Not trivial, would probably be Hawaii, but that would be winning a lot of oceanic area.

7.2 Something else?

8 Conclusion

We have shown the design of two algorithms in the field of computational geometry. We found that altering the the Fortune algorithm to return a fast search data structure of a trapezoidal map for point location was indeed feasible, and showed that by querying random points at the datastructure we could reason about which cell is the biggest. The algorithms presented are implemented in a generic fashion and it should be easy to see the solution used on top of another any other data set.

Acknowledgements We would like to thank Rasmus Pagh for his lectures and his on the spot guidance that has been essential for moving this project in the right direction.

References

[1] O. Schwarzkopf, M. Overmars, M. van Kreveld, and M. de Berg. *Computational Geometry, Algorithms and Applications*. Springer, 1997.