A Sweep Line Algorithm for Nearest Neighbour Queries

João Dinis
Departamento de Optoelectrónica
Instituto Nacional de Engenharia e Tecnologia Industrial
Estrada do Paço do Lumiar
1649-038 Lisboa, Portugal
jdinis@dop.ineti.pt

Margarida Mamede
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
mm@di.fct.unl.pt

Abstract

We introduce a novel algorithm for solving the nearest neighbour problem when the query points are known in advance, which is based on Fortune's plane sweep algorithm. The crucial idea is to use the wavefront for solving the nearest neighbour queries as the Voronoi diagram is being computed, instead of storing it in an auxiliary data structure, as the algorithm presented by Lee and Yang [9] does, and then querying that data structure.

Although our algorithm is not optimal in terms of its worst-case behaviour, it runs in $O(m \log m)$ expected time, where m is the total number of points (sites and query points). Experimental results show that it outperforms the algorithm of Lee and Yang, provided the number of query points does not exceed four times the number of sites.

1 Introduction

The family of point set pattern matching problems has been widely studied in recent years [1, 3, 7, 8], due to the variety of fields where these problems are applied. As in [11], we tackle the problem of star mapping, where points belong to the $\mathbb{R}^2$ Euclidean space, having adopted the alignment method introduced by Goodrich *et al.* [8]. Basically, the matching test consists in solving a nearest neighbour query for each point of the pattern in the appropriate subset of the background, for which the Voronoi diagram is computed.

The novel idea is to solve all nearest neighbour queries at the same time that the Voronoi diagram is computed, instead of storing it in an auxiliary static data structure, such as a trapezoidal map, and then querying that data structure [10]. This may be done because all query points (the pattern) are known in advance.

Although the algorithm introduced by Lee and Yang [9] makes the same requirement, as it works on a planar subdivision, it performs two steps: in the first one, the planar subdivision is built and, in the second, the nearest neighbour points are identified.

Both approaches rely on the plane sweep technique (due to Fortune [6]). However, the algorithm of Lee and Yang solves the batched point location problem in a planar subdivision, whereas ours solves the batched point location problem in a set of points. It is worth mentioning that, in [5], where Edelsbrunner and Overmars discuss some techniques for coping with the more general problem of batched searching, the authors argue that the plane sweep technique is an appropriate choice for 2-dimensional batched search problems.

The rest of the paper is organized as follows. Section 2 describes our approach, Section 3 studies its time and space complexity, Section 4 presents some experimental results that compare the performance of both algorithms, and Section 5 includes some comments on the research done in the paper. Detailed proofs can be found in [4].

2 The Algorithm

Our approach relies on the plane sweep technique (due to Fortune [6]), which computes a Voronoi diagram with n point sites in $O(n \log n)$ time, in the worst case. Intuitively, a horizontal line, denoted by *sweep line*, sweeps the plane from the top to the bottom. There is another line, called the *wavefront*, which is made up of parabolic arcs that are defined by the sweep line and the point sites on or above it.

The key fact is that, at any time, the points that lie on a parabolic arc of the wavefront, generated by a site p, are necessarily at least as close to p as to any other site. Moreover, during the sweep process, the parabolic arcs generated by p scan all points of the plane closer to p than to any other site. Therefore, for every query point

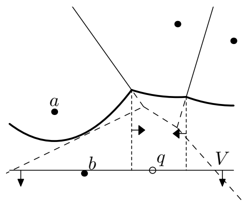

Figure 1: Query-event.

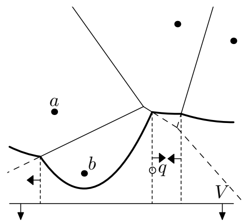

Figure 2: Intersection-event.

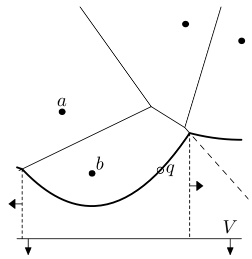

Figure 3: Arc-event.

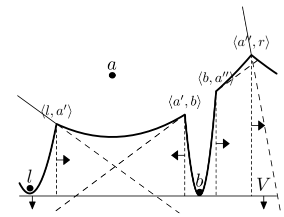

Figure 4: Site-event.

q, it is enough to determine which parabolic arc $\langle a \rangle$ of the wavefront contains q. The site that generated $\langle a \rangle$ is the nearest neighbour of q. The wavefront scans the query point once, because the sweep is y-monotonic, and the point is contained in only one of its elements, because the wavefront is x-monotonic.

The wavefront can be seen as a sequence of alternate parabolic arcs and intersections of parabolic arcs. Let us consider the regions bounded by an arc (on the top), the sweep line (on the bottom), and the vertical extensions of the two intersections adjacent to the arc (Figs. 1–5). Along the sweep, the regions' shape changes: the arc moves downwards; and each intersection traces out an edge of the Voronoi diagram, moving on either x-monotonously to the left, or x-monotonously to the right, or vertically (i.e., keeping the x-coordinate constant).

Let us call the event where a new query point is reached by the sweep line a *query-event* (Fig. 1). When a query-event takes place, we do not know which parabolic arc will reach the point, since the nearest site can be located below it, so the arc may not even exist. However, it is easy to find the region that contains it, by performing a search similar to the one made for a *site-event*. When the sweep proceeds, the algorithm keeps tracking the area where the query point lies, possibly moving the point into another region, until it is reached by an arc.

The procedure to handle a query-event associated with a query point q is the following.

1. Determine the region that contains q.
2. Determine which of the following three situations occurs first: the arc reaches q, which is an *arc-event*; the left vertical line reaches q, which is an *intersection-event*; or the right vertical line reaches q, which is also an *intersection-event*. Associate q with the element (the arc or the intersection) of the event.
3. Insert the event in the event priority queue.

Remark that, besides associating the query point with an element of the wavefront, this procedure generates an event to be handled later.

To handle an intersection-event (Fig. 2) is to change the element of the wavefront which the query point is associated with. The region that now contains the point is either the one on the left or the one on the right, depending on the x-direction of the intersection (which is assured to be either from left to right or from right to left). The movement direction of an intersection adjacent to the sites $\langle p, p' \rangle$ is (where $p_x \leq p'_x$):

- from left to right if $p_y < p'_y$;
- from right to left if $p_y > p'_y$; and
- downwards if $p_y = p'_y$.

Then, the algorithm must determine, in the new region, which event will first take place, and the choice is between the new arc and the vertical line on the opposite side of the intersection-event.

This procedure is repeated until an arc-event is handled (Fig. 3). At that moment, the point has been reached by a parabolic arc, thus the nearest site has been found.

Let us now explain how the sweep line position is computed at these new types of events.

For arc-events, it follows from the definition of parabola that an arc generated by a site p contains a point q when the distance between q and the sweep line, $q_y - V_y$, is equal to the distance between q and p: $\text{distance}(q, p) = q_y - V_y$.

For an intersection-event adjacent to two sites p_1 and p_2, the algorithm computes the bisector of p_1 and p_2, which defines the positions of the intersection along the sweep. Let $q = (q_x, q_y)$ be the query point coordinates, and $b = (q_x, b_y)$ be the point of the bisector with the same x-coordinate. When the intersection reaches b, the vertical line contains q. So, the event should take place when the distance between b and the sweep line, $b_y - V_y$, is equal to the distance between b and p_1 (or p_2): $\text{distance}(b, p_1) = b_y - V_y$.

Notice that, so far, a query point is always associated with the arc or an intersection of the region containing it, depending on which reaches it first. As we shall see, this condition (called the *invariant on the query* *points*) always holds. In addition, for a query point q to be moved from one region r_1 to another region r_2, the site that gives rise to the arc of r_2 is closer to q than the site associated with r_1.

Like Fortune's algorithm [2], our algorithm makes use of two data structures: a balanced binary tree, to store the arcs and the intersections of the wavefront; and a priority queue of events. As sites and query points are known in advance, all site-events and query-events are generated and inserted in the queue, in the beginning.

For the sake of efficiency, arc-events and intersection-events store a pointer to the corresponding tree node (arc or intersection). Furthermore, every node of the binary tree (internal or leaf) stores a linked list of query points, and every element of the list has a reference to the corresponding event in the priority queue.

A site-event creates a new region in the middle of some region. Thus, the query points associated with that region have to be distributed among the three new regions (Fig. 4). Let then $[\langle l,a\rangle, a, \langle a,r\rangle]$ be the existing region, bounded by the left intersection $\langle l,a\rangle$, the arc $\langle a\rangle$, and the right intersection $\langle a,r\rangle$. The introduction of a new arc $\langle b\rangle$ gives rise to three regions: $[\langle l,a'\rangle, a', \langle a',b\rangle]$, $[\langle a',b\rangle, b, \langle b,a''\rangle]$, and $[\langle b,a''\rangle, a'', \langle a'',r\rangle]$. The algorithm performs the following steps.

1. The events of arc $\langle a\rangle$ are distributed among the arcs $\langle a'\rangle$, $\langle b\rangle$, and $\langle a''\rangle$, based on the x-coordinate of the query points.

2. For every arc-event of $\langle a'\rangle$ (resp., $\langle a''\rangle$), the algorithm checks whether the corresponding query point is first reached by the intersection $\langle a',b\rangle$ (resp., $\langle b,a''\rangle$) and, if that is the case, the arc-event becomes an intersection-event associated with that intersection.

3. If the intersection $\langle l,a'\rangle$ is moving on to the right (resp., if the intersection $\langle a'',r\rangle$ is moving on to the left), its events are distributed among itself, $\langle a',b\rangle$, $\langle b\rangle$, and $\langle b,a''\rangle$.

Needless to say, the priority queue is updated whenever there is a change in the lists of query points. Besides, the invariant on the query points is kept.

Remark that, if a site p occurs vertically below a query point q, q gives rise to an event associated with the new arc $\langle b\rangle$, because $\langle b\rangle$ is a vertical line segment that contains q. The priority of that arc-event, which corresponds to the current position of the wavefront, is highest than the priority of any event in the queue. Consequently, the event will be handled next, ending with the conclusion that p is the site nearest to q.

A circle-event (Fig. 5) corresponds to the joining of two intersections, where an arc drops out, the two edges scanned by the intersections meet (defining a vertex of the Voronoi diagram), and a new edge starts.

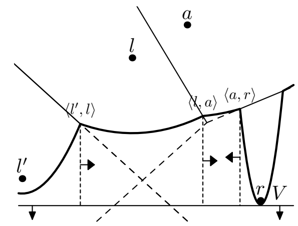

Figure 5: Circle-event.

By construction, no query point can be associated with the arc $\langle a\rangle$ when it disappears, or with the intersections $\langle l,a\rangle$ and $\langle a,r\rangle$ unless they have been moving on the same direction. Actually, if $\langle l,a\rangle$ and $\langle a,r\rangle$ have been moving on opposite directions, their lists of query points are empty because they have scanned all query points that could be associated with them.

In any case, let $[\langle l,a\rangle, a, \langle a,r\rangle]$ be the region that disappears, and $[\langle l',l\rangle, l, \langle l,a\rangle]$ and $[\langle a,r\rangle, r, \langle r,r'\rangle]$ be its adjacent regions. The event distribution is as follows.

1. The intersection-events of $\langle l,a\rangle$ and $\langle a,r\rangle$, if any, become associated with the new intersection $\langle l,r\rangle$.

2. If $\langle l,r\rangle$ moves to the right, determine, for every event of $\langle r\rangle$ and for every event of $\langle r,r'\rangle$, whether it is first reached by $\langle l,r\rangle$ and, in that case, associate it with the intersection $\langle l,r\rangle$.

3. If $\langle l,r\rangle$ moves to the left, determine, for every event of $\langle l\rangle$ and for every event of $\langle l',l\rangle$, whether it is first reached by $\langle l,r\rangle$ and, in that case, associate it with the intersection $\langle l,r\rangle$.

Once again, this assures (c.f. [4]) that the invariant condition on the query points remains true for the two new regions $[\langle l',l\rangle, l, \langle l,r\rangle]$ and $[\langle l,r\rangle, r, \langle r,r'\rangle]$.

There are two special cases that deserve attention. To overcome the problem caused by query points above the first site, which would be reached by the sweep line before the first region had been created, the sweep process starts only with the first site-event. At that moment, the first arc is created, and all query points located above the site are associated with it. The second case is when a query point q occurs on the vertical extension of an intersection. If the intersection moves on to the left or to the right, the corresponding intersection-event will be handled without delay, and q will become associated with another event. But if the intersection scans a vertical edge of the Voronoi diagram, q cannot remain associated with it, because the treatment of that event would not cause any change in the system. Therefore, in this case, q is associated with one of the joining arcs.

3 Time and Space Complexity

The next goal is to analyse the cost of our algorithm with n sites and k query points.

Apart from a list of query points in each node, the binary tree is the same as that of Fortune's algorithm. Hence, it has $O(n)$ nodes and each search, insertion or removal operation takes $O(\log n)$ steps. As for the priority queue, since it stores exactly the same site-events and circle-events, plus at most one event per query point, its length is $O(n+k)$ and each operation on it costs $O(\log(n+k))$ time. In what concerns memory requirements, both data structures use $O(n+k)$ space.

Leaving out, for now, the operations respecting to the query points, the processing of site-events and circle-events remains unchanged. So, there are $\Theta(n)$ of these events, which take $O(n \log(n+k))$ time to process, because of the queue length. Concerning query points, it is easy to verify that:

- the time spent in creating, removing, or handling any query-event, intersection-event, or arc-event is within $O(\log(n+k))$; and
- when site-events and circle-events are processed, the time required to visit each element of a list of query points and to decide whether to reschedule or not the corresponding event is constant.

So, our next step is to estimate how many events on query points are generated and how many visits to query points are performed.

Let then q be an arbitrary query point, and e be an intersection-event or an arc-event associated with q. We say that *e links a site p with q* (or, alternatively, that a site *p is linked with q through e*), if p is the site of the region that contains q when e is generated. Although we might consider that a query point belongs to two regions when an intersection-event is handled, we shall assume, according to the intuitions spelled out above, that it is already in the new one (instead of in the region that contains it at the time the intersection-event is generated).

This notion of link is extended to visits: *a visit v to q links a site p with q* (or *p is linked with q through v*), if q belongs to a region bounded by an arc generated by p by the time the visit v is performed.

The sequence of events and visits related to q can be split into sub-sequences according to the linked site. More concretely, it can be seen as having the following structure:

$$e' \quad \underbrace{e_1\ x_{11}\ \cdots\ x_{1l_1}}_{p_1} \quad \cdots \quad \underbrace{e_u\ x_{u1}\ \cdots\ x_{ul_u}}_{p_u}$$

where e' is the query-event, which does not link any site with q, e_i represents the first event that links site p_i with q, and x_{ij} stands for an event or a visit through which p_i is linked with q. In particular, e_1 is the intersection-event or arc-event generated by the procedure for handling query-events, whereas x_{ul_u} is the only arc-event that takes place.

Let us first concentrate on the events e_i. Remark that, for the site linked with q to change, either an intersection-event is handled, or a site occurs vertically below q. In both cases, q is closer to the new site than to the old one, which allows us to conclude that the sites $p_1, p_2, \ldots, p_u$ linked with q are all distinct.

Moreover, it can be proved [4] that, for every site p_i linked with q, there is a circle whose boundary contains p_i and q, and that does not contain any site in its interior. Therefore, the total number of sites that may be linked with q (i.e., the value of u) cannot exceed the number of neighbours of q in the Voronoi diagram of $P \cup \{q\}$, where P stands for the set of sites. This implies, together with the properties on Voronoi polygons [2], that, even though u can be as large as n, in some rare cases, its expected value is $O(1)$.

In what concerns the events x_{ij} (for some fixed i), notice that they cannot be generated when a query-event, intersection-event or arc-event is handled. But, for every site-event that affects a region associated with p_i, q is visited once and at most one new event is generated (as, in practice, the first two steps are performed simultaneously). Besides, a new edge of the Voronoi polygon of p_i is created. The same happens with circle-events. That is, whenever a visit (and the possible corresponding event) links p_i with q in the context of a circle-event, a new edge of the Voronoi polygon of p_i starts to be traced out. This means that the number of such visits (and events) cannot exceed the number of edges of the Voronoi polygon of p_i. Once more, although it is well-known that a single Voronoi polygon may have $n-1$ edges, the average number of edges of the Voronoi polygons is less than six [2].

We conclude that the time spent due to the query points is $\Theta(nk \log(n+k))$, in the worst-case, and $O(k \log(n+k))$, in the average-case. Hence, the algorithm uses $O(n+k)$ space, and takes $\Theta(nk \log(n+k))$ worst-case time and $O((n+k) \log(n+k))$ expected time. The runtime bound is expected over random site positions.

4 Experimental Results

We present some experimental results that compare the performance of our algorithm and that of Lee and Yang, which runs in $O((n+k) \log(n+k))$ time and also re-

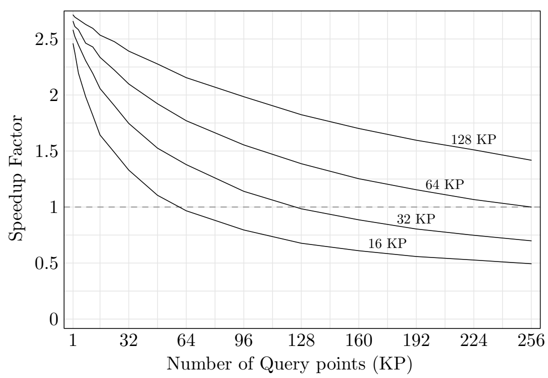

Figure 6: Lee and Yang / sweep line running time. Each line refers to a set of sites.

quires that the query points be known in advance [9]. Both algorithms rely on the same two data structures, which have been implemented with a red-black tree and a binary heap.

We measured the running times with four sets of sites, with 16, 32, 64, and 128 KP (where 1 KP stands for 1024 points). For each one of them, we used sixteen sets of query points, with 1, 2, 4, 8, 12, 16, 24, 32, 48, 64, 96, 128, 160, 192, 224 and 256 KP. The measurements were made on a 400 MHz Pentium II processor with a 256 MB RAM.

Fig. 6 compares both sets of results. Each line represents the quotient between the running time of the algorithm of Lee and Yang and the running time of our algorithm. So, our algorithm outperforms the other when the curves are above the dashed line.

It turns out that our algorithm is the fastest when the number of query points does not exceed four times the number of sites. ¿From then on, the algorithm of Lee and Yang outperforms ours. The justification is that the high cost of building the binary search tree they use to solve the nearest neighbour queries is amortized as the number of query points grows. Furthermore, in all experiments made, the running time of our algorithm never exceeded twice the running time of the algorithm by Lee and Yang.

5 Conclusions

We have presented an algorithm to solve the offline nearest neighbour query problem, which runs in $O(m \log m)$ expected time, where m is the total number of points (sites and query points).

We conclude that our algorithm constitutes an alternative to the algorithm of Lee and Yang. The choice of which to use depends on the characteristics of the application and, in particular, on the relation between the number of query points and the number of sites. In our case, since those numbers are of a similar magnitude, our algorithm is the best suited for the job.

References

[1] H. Alt and L. J. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 121–153. Elsevier, 2000.

[2] F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier, 2000.

[3] L. P. Chew, M. T. Goodrich, D. P. Huttenlocher, K. Kedem, J. M. Kleinberg, and D. Kravets. Geometric pattern matching under Euclidean motion. *Comptutational Geometry: Theory and Applications*, 7:113–124, 1997.

[4] J. Dinis and M. Mamede. A sweep line algorithm for multiple nearest neighbour queries. Technical Report RT 1/2002, Universidade Nova de Lisboa, April 2002. http://ctp.di.fct.unl.pt/˜mm/sweep1.ps.gz.

[5] H. Edelsbrunner and M. H. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6(4):515–542, 1985.

[6] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[7] M. Gavrilov, P. Indyk, R. Motwani, and S. Venkatasubramanian. Geometric pattern matching: A performance study. In *Proc. of the 15th Annual ACM Symposium on Computational Geometry*, pages 79–85, 1999.

[8] M. T. Goodrich, J. S. B. Mitchell, and M. W. Orletsky. Approximate geometric pattern matching under rigid motions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(4):371–379, 1999.

[9] D. T. Lee and C. C. Yang. Location of multiple points in a planar subdivision. *Information Processing Letters*, 9(4):190–193, 1979.

[10] F. P. Preparata. Planar point location revisited. *International Journal of Foundations of Computer Science*, 1(1):71–86, 1990.

[11] G. Weber, L. Knipping, and H. Alt. An application of point pattern matching in astronautics. *Journal of Symbolic Computation*, 11:1–20, 1994.