OUTPUT-SENSITIVE ALGORITHMS FOR COMPUTING NEAREST-NEIGHBOUR DECISION BOUNDARIES*

David Bremner† Erik Demaine‡ Jeff Erickson§ John Iacono¶ Stefan Langerman‖
Pat Morin** Godfried Toussaint††

ABSTRACT. Given a set R of red points and a set B of blue points, the *nearest-neighbour decision rule* classifies a new point q as red (respectively, blue) if the closest point to q in $R \cup B$ comes from R (respectively, B). This rule implicitly partitions space into a red set and a blue set that are separated by a red-blue *decision boundary*. In this paper we develop output-sensitive algorithms for computing this decision boundary for point sets on the line and in $\mathbb{R}^2$. Both algorithms run in time $O(n \log k)$, where k is the number of points that contribute to the decision boundary. This running time is the best possible when parameterizing with respect to n and k.

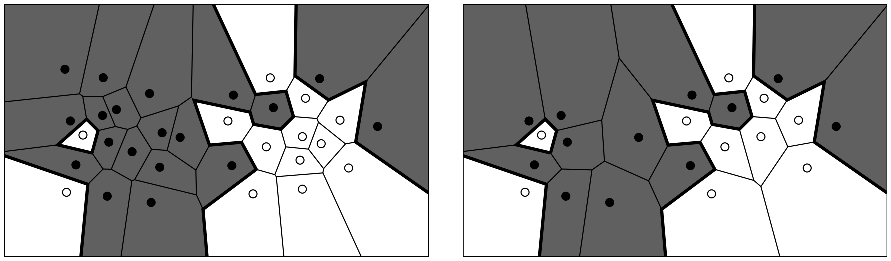
1 Introduction

Let S be a set of n points in the plane that is partitioned into a set of *red points* denoted by R and a set of *blue points* denoted by B. The *nearest-neighbour decision rule* classifies a new point q as the colour of the closest point to q in S. The nearest-neighbour decision rule is popular in pattern recognition as a means of learning by example. For this reason, the set S is often referred to as a *training set*.

Several properties make the nearest-neighbour decision rule quite attractive, including its intuitive simplicity and the theorem that the asymptotic error rate of the nearest-neighbour rule is bounded from above by twice the Bayes error rate [6, 8, 16]. (See [17] for an extensive survey of the nearest-neighbour decision rule and its relatives.) Furthermore, for point sets in small dimensions, there are efficient and practical algorithms for preprocessing a set S so that the nearest neighbour of a query point q can be found quickly.

The nearest-neighbour decision rule implicitly partitions the plane into a red set and a blue set that meet at a red-blue *decision boundary*. One attractive aspect of the nearest-neighbour decision rule is that it is often possible to reduce the size of the training set S without changing the decision boundary. To see this, consider the *Voronoĭ diagram* of S, which partitions the plane into convex (possibly unbounded) polygonal *Voronoĭ cells*, where the Voronoĭ cell of point $p \in S$ is the set of all points that are closer to p

*This research was partly funded by the Alexander von Humboldt Foundation and The Natural Sciences and Engineering Research Council of Canada.
†Faculty of Computer Science, University of New Brunswick, bremner@unb.ca
‡MIT Laboratory for Computer Science, edemaine@mit.edu
§Computer Science Department, University of Illinois, jeffe@cs.uiuc.edu
¶Polytechnic University, jiacono@poly.edu
‖Chargé de recherches du FNRS, Université Libre de Bruxelles, stefan.langerman@ulb.ac.be
**School of Computer Science, Carleton University, morin@cs.carleton.ca
††School of Computer Science, McGill University, godfried@cs.mcgill.ca



(a) (b)

Figure 1: The Voronoĭ diagram (a) before Voronoĭ condensing and (b) after Voronoĭ condensing. Note that the decision boundary (in bold) is unaffected by Voronoĭ condensing. Note: In this figure, and all other figures, red points are denoted by white circles and blue points are denoted by black disks.

than to any other point in S (see Figure 1.a). If the Voronoĭ cell of a red point r is completely surrounded by the Voronoĭ cells of other red points then the point r can be removed from S and this will not change the classification of any point in the plane (see Figure 1.b). We say that these points *do not contribute* to the decision boundary, and the remaining points *contribute* to the decision boundary.

The preceding discussion suggests that one approach to reducing the size of the training set S is to simply compute the Voronoĭ diagram of S and remove any points of S whose Voronoĭ cells are surrounded by Voronoĭ cells of the same colour. Indeed, this method is referred to as *Voronoĭ condensing* [18]. There are several $O(n \log n)$ time algorithms for computing the Voronoĭ diagram a set of points in the plane, so Voronoĭ condensing can be implemented to run in $O(n \log n)$ time.[1] However, in this paper we show that we can do significantly better when the number of points that contribute to the decision boundary is small. Indeed, we show how to do Voronoĭ condensing in $O(n \log k)$ time, where k is the number of points that contribute to the decision boundary (i.e., the number of points of S that remain after Voronoĭ condensing). We also show that the same result holds even if there are $c > 2$ colour classes. Algorithms, like these, in which the size of the input and the size of the output play a role in the running time are referred to as *output-sensitive* algorithms.

Readers familiar with the literature on output-sensitive convex hull algorithms may recognize the expression $O(n \log k)$ as the running time of optimal algorithms for computing convex hulls of n point sets with k extreme points, in 2 or 3 dimensions [2, 4, 5, 13, 19]. This is no coincidence. Given a set of n points in $\mathbb{R}^2$, we can colour them all red and add three blue points at infinity (see Figure 2). In this set, the only points that contribute to the nearest-neighbour decision boundary are the three blue points and the red points on the convex hull of the original set. Thus, identifying the points that contribute to the nearest-neighbour decision boundary is at least as difficult as computing the extreme points of a set.

Observe that, once the size of the training set has been reduced by Voronoĭ condensing, the condensed set can be preprocessed in $O(k \log k)$ time to answer nearest neighbour queries in $O(\log k)$ time per query. This makes it possible to do nearest-neighbour classifications in $O(\log k)$ time. Alternatively, the algorithm we describe for computing the nearest neighbour decision boundary actually produces the Voronoĭ diagram of the condensed set (which has size $O(k)$) that can be preprocessed in $O(k)$ time by Kirkpatrick's point-location algorithm [12] to allow nearest neighbour classification in $O(\log k)$ time.

[1] Historically, the first efficient algorithm for specifically computing the nearest-neighbour decision boundary is due to Dasarathy and White [7] and runs in $O(n^4)$ time. The first $O(n \log n)$ time algorithm for computing the Voronoĭ diagram of a set of n points in the plane is due to Shamos [15].

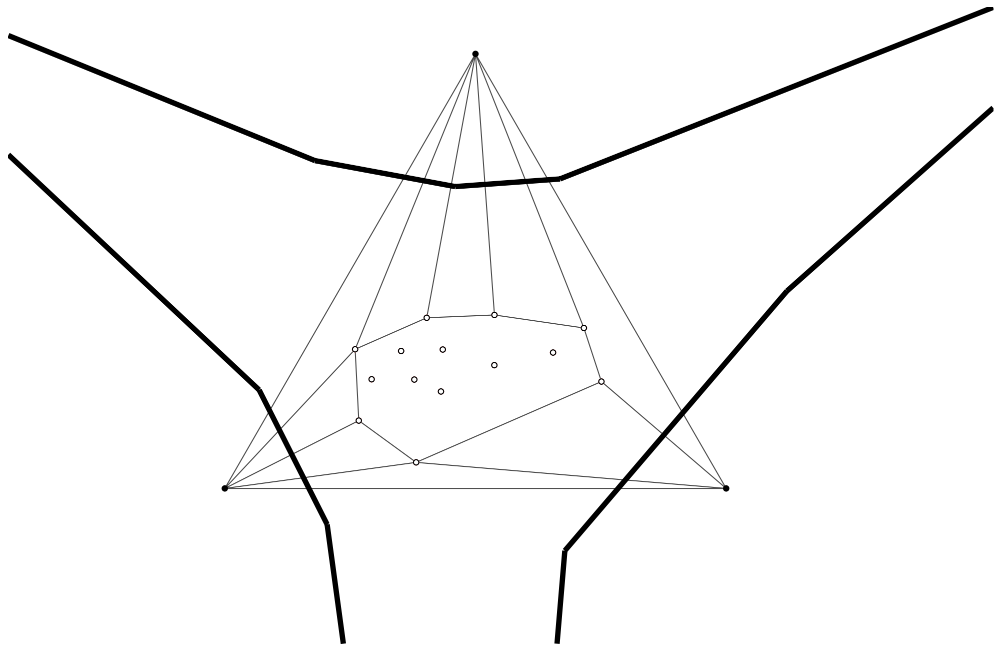


Figure 2: The relationship between convex hulls and decision boundaries. Each vertex of the convex hull of R contributes to the decision boundary.

The remainder of this paper is organized as follows: In Section 2 we describe an algorithm for computing the nearest-neighbour decision boundary of points on a line that runs in $O(n \log k)$ time. In Section 3 we present an algorithm for points in the plane that also runs in $O(n \log k)$ time. Finally, in Section 4 we summarize and conclude with open problems.

2 A 1-Dimensional Algorithm

In the 1-dimensional version of the nearest-neighbour decision boundary problem, the input set S consists of n real numbers. Imagine sorting S, so that $S = \{s_1, \ldots, s_n\}$ where $s_i < s_{i+1}$ for all $1 \leq i < n$. The decision boundary consists of all pairs (s_i, s_{i+1}) where s_i is red and s_{i+1} is blue, or *vice-versa*. Thus, this problem is solvable in linear-time if the points of S are sorted. Since sorting the elements of S can be done using any number of $O(n \log n)$ time sorting algorithms, this immediately implies an $O(n \log n)$ time algorithm. Next, we give an algorithm that runs in $O(n \log k)$ time and is similar in spirit to Hoare's quicksort [11].

To find the decision boundary in $O(n \log k)$ time, we begin by computing the median element $m = s_{\lceil n/2 \rceil}$ in $O(n)$ time using any one of the existing linear-time median finding algorithms (see [3]). Using an additional $O(n)$ time, we split S into the sets $S_1 = \{s_1, \ldots, s_{\lceil n/2 \rceil - 1}\}$ and $S_2 = \{s_{\lceil n/2 \rceil + 1}, \ldots, s_n\}$ by comparing each element of S to the median element m. At the same time we also find $s_{\lceil n/2 \rceil - 1}$ and $s_{\lceil n/2 \rceil + 1}$ by finding the maximum and minimum elements of S_1 and S_2, respectively. We then check if $(s_{\lceil n/2 \rceil - 1}, m)$ and/or $(m, s_{\lceil n/2 \rceil + 1})$ are part of the decision boundary and report them if necessary.

At this point, a standard divide-and-conquer algorithm would recurse on both S_1 and S_2 to give an $O(n \log n)$ time algorithm. However, we can improve on this by observing that it is not necessary to recurse on a subproblem if it contains only elements of one colour, since it will not contribute a pair to the decision boundary. Therefore, we recurse on each of S_1 and S_2 only if they contain at least one red element and one blue element.

The correctness of the above algorithm is clear. To analyze its running time we observe that the running time is bounded by the recurrence

$$T(n, k) \leq O(n) + T(n/2, l) + T(n/2, k - l) \quad ,$$

where l is the number of points that contribute to the decision boundary in S_1 and where $T(1, k) = O(1)$ and $T(n, 0) = O(n)$. An easy inductive argument that uses the concavity of the logarithm shows that this recurrence is maximized when $l = k/2$, in which case the recurrence solves to $O(n \log k)$ [5].

Theorem 1. *The nearest-neighbour decision boundary of a set of n red and blue real numbers can be computed in $O(n \log k)$ time, where k is the number of elements that contribute to the decision boundary.*

3 A 2-Dimensional Algorithm

In the 2-dimensional nearest-neighbour decision boundary problem the Voronoĭ cells of S are (possibly unbounded) convex polygons and the goal is to find all Voronoĭ edges that bound two cells whose defining points have different colours. Throughout this section we will assume that the points of S are in *general position* so that no four points of S lie on a common circle. This assumption is not very restrictive, since general position can be simulated using infinitesmal perturbations of the input points.

It will be more convenient to present our algorithm using the terminology of Delaunay triangulations. A *Delaunay triangle* in S is a triangle whose vertices (v_1, v_2, v_3) are in S and such that the circle with v_1, v_2 and v_3 on its boundary does not contain any point of S in its interior. A *Delaunay triangulation* of S is a partitioning of the convex hull of S into Delaunay triangles. Alternatively, a *Delaunay edge* is a line segment whose vertices (v_1, v_2) are in S and such that there exists a circle with v_1 and v_2 on its boundary that does not contain any point of S in its interior. When S is in general position, the Delaunay triangulation of S is unique and contains all triangles whose edges are Delaunay edges (see [14]). It is well known that the Delaunay triangulation and the Voronoĭ diagram are dual in the sense that two points of S are joined by an edge in the Delaunay triangulation if and only if their Voronoĭ cells share an edge.

We call a Delaunay triangle or Delaunay edge *bichromatic* if its set of defining vertices contains at least one red and at least one blue point of S. Thus, the problem of computing the nearest-neighbour decision boundary is equivalent to the problem of finding all bichromatic Delaunay edges.

3.1 The High Level Algorithm

In the next few sections, we will describe an algorithm that, given a value $\kappa \geq k$, finds the set of all bichromatic Delaunay triangles in S in $O((\kappa^2 + n) \log \kappa)$ time, which for $\kappa \leq \sqrt{n}$ simplifies to $O(n \log \kappa)$. To obtain an algorithm that runs in $O(n \log k)$ time, we repeatedly guess the value of κ, run the algorithm

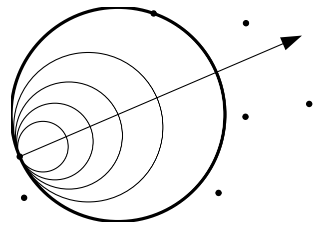


Figure 3: A pivot operation.

until we find the entire decision boundary or until it determines that $\kappa < k$ and, in the latter case, restart the algorithm with a larger value of κ. If we ever reach a point where the value of κ exceeds $\sqrt{n}$ then we stop the entire algorithm and run an $O(n \log n)$ time algorithm to compute the entire Delaunay triangulation of S.

The values of κ that we use are $\kappa = 2^{2^i}$ for $i = 0, 1, 2, \ldots, \lceil \log \log n \rceil$. Since the algorithm will terminate once $\kappa \geq k$ or $\kappa \geq \sqrt{n}$, the total cost of all runs of the algorithm is therefore

$$T(n,k) = \sum_{i=0}^{\lceil \log \log k \rceil} O(n \log 2^{2^i}) = \sum_{i=0}^{\lceil \log \log k \rceil} O(n 2^i) = O(n \log k) \ ,$$

as required.

3.2 Pivots

A key subroutine in our algorithm is the *pivot*[2] operation illustrated in Figure 3. A pivot in the set of points S takes as input a ray and reports the largest circle whose center is on the ray, has the origin of the ray on its boundary and has no point of S in its interior. We will make use of the following data structuring result, due to Chan [4]. For completeness, we also include a proof.

Lemma 1 (Chan 1996). *Let S be a set of n points in $\mathbb{R}^2$. Then, for any integer $1 \leq m \leq n$, there exists a data structure of size $O(n)$ that can be constructed in $O(n \log m)$ time, and that can perform pivots in S in $O(\frac{n}{m} \log m)$ time per pivot.*

Proof. Dobkin and Kirkpatrick [9, 10] show how to preprocess a set S of n points in $O(n \log n)$ time to answer pivot queries in $O(\log n)$ time per query. Chan's data structure simply partitions S into n/m groups each of size m and then uses the Dobkin-Kirkpatrick data structure on each group. The time to build all n/m data structures is $\frac{n}{m} \times O(m \log m) = O(n \log m)$. To perform a query, we simply query each of the n/m data structures in $O(\log m)$ time per data structure and report the smallest circle found, for a query time of $\frac{n}{m} \times O(\log m) = O(\frac{n}{m} \log m)$. $\square$

In the following, we will be using Lemma 1 with a value of $m = \kappa^2$, so that the time to construct the data structure is $O(n \log \kappa)$ and the query time is $O(\frac{n}{\kappa^2} \log \kappa)$. We will use two such data structures,

[2] The term pivot comes from linear programming. The relationship between a (polar dual) linear programming pivot and the circular pivot described here is evident when we consider the parabolic lifting that transforms the problem of computing a 2-dimensional Delaunay triangulation to that of computing a 3-dimensional convex hull of a set of points on the paraboloid $z = x^2 + y^2$. In this case, the circle is the projection of the intersection of a plane with the paraboloid.

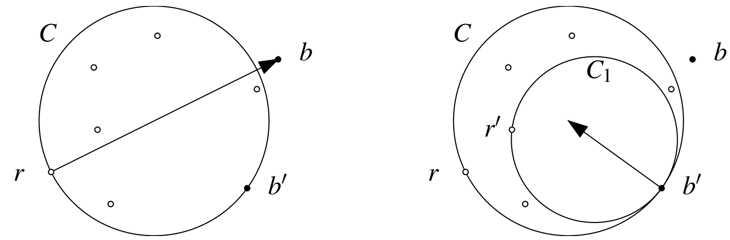


Figure 4: The (a) first and (b) second pivot used to find a bichromatic edge (r', b').

one for performing pivots in the set R of red points and one for performing pivots in the set B of blue points.

3.3 Finding the First Edge

The first step in our algorithm is to find a single bichromatic edge of the Delaunay triangulation. Refer to Figure 4. To do this, we begin by choosing any red point r and any blue point b. We then perform a pivot in the set B along the ray with origin r that contains b. This gives us a circle C that has no blue points in its interior and has r as well as some blue point b' (possibly $b = b'$) on its boundary. Next, we perform a pivot in the set R along the ray originating at b' and passing through the center of C. This gives us a circle C_1 that has no point of S in its interior and has b' and some red point r' (possibly $r = r'$) on its boundary. Therefore, (r', b') is a bichromatic edge in the Delaunay triangulation of S.

The above argument shows how to find a bichromatic Delaunay edge using only 2 pivots, one in R and one in B. The second part of the argument also implies the following useful lemma.

Lemma 2. *If there is a circle with a red point r and a blue point b on its boundary, and no red (respectively, blue) points in its interior, then r (respectively, b) contributes to the decision boundary.*

3.4 Finding More Points

Let Q be the set of points that contribute to the decision boundary, i.e., the set of points that are the vertices of bichromatic triangles in the Delaunay triangulation of S. Suppose that we have already found a set $P \subseteq Q$ and we wish to either (1) find a new point $p \in Q \setminus P$ or (2) verify that $P = Q$.

To do this, we will make use of the *augmented Delaunay triangulation* of P (see Figure 5). This is the Delaunay triangulation of $P \cup \{v_1, v_2, v_3\}$, where v_1, v_2, and v_3 are three *black* points "at infinity" (see Figure 5). For any triangle t, we use the notation $C(t)$ to denote the circle whose boundary contains the three vertices of t (note that if t contains a black point then $C(t)$ is a halfplane). The following lemma allows us to tell when we have found the entire set of points Q that contribute to the decision boundary.

Lemma 3. *Let $\emptyset \neq P \subseteq Q$. The following statements are equivalent:*

1. *For every triangle t in the augmented Delaunay triangulation of P, if t has a blue (respectively, red)*

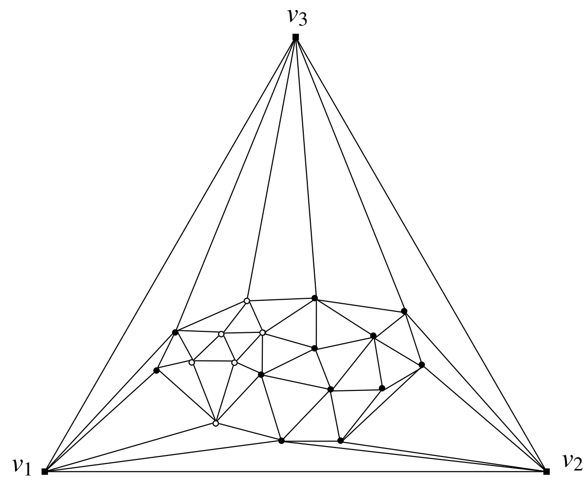


Figure 5: The augmented Delaunay triangulation of S.

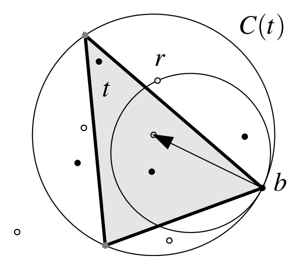


Figure 6: If Statement 1 of Lemma 3 is not true then $P \neq Q$.

vertex then $C(t)$ does not have a red (respectively, blue) point of S in its interior.

2. $P = Q$.

Proof. First we show that if Statement 1 of the lemma is not true, then Statement 2 is also not true, i.e., $P \neq Q$. Suppose there is some triangle t in the augmented Delaunay triangulation of P such that t has a blue vertex b and $C(t)$ contains a red point of S in its interior. Pivot in R along the ray originating at b and passing through the center of $C(t)$ (see Figure 6). This will give a circle C with b and some red point $r \notin P$ on its boundary and with no red points in its interior. Therefore, by Lemma 2, r contributes to the decision boundary and is therefore in Q, so $P \neq Q$. A symmetric argument applies when t has a red vertex r and $C(t)$ contains a blue vertex in its interior.

Next we show that if Statement 2 of the lemma is not true then Statement 1 is not true. Suppose that $P \neq Q$. Let r be a point in $Q \setminus P$ and, without loss of generality, assume r is a red point. Since $r \in Q$, there is a circle C with r and some other blue point b on its boundary and with no points of S in its interior. We will use r and b to show that the augmented Delaunay triangulation of P contains a triangle t such that either (1) b is a vertex of t and $C(t)$ contains r in its interior, or (2) $C(t)$ contains both r and b in its interior. In either case, Statement 1 of the lemma is not true because of triangle t.

Refer to Figure 7 for what follows. Consider the largest circle C_1 that is concentric with C and that contains no point of P in its interior (this circle is at least as large as C). The circle C_1 will have at least one point p_1 of P on its boundary (it could be that $p_1 = b$, if $b \in P$). Next, perform a pivot in P along the ray originating at p_1 and containing the center of C_1. This will give a circle C_2 that contains C_1 and with two points p_1 and p_2 of $P \cup \{v_1, v_2, v_3\}$ on its boundary and with no points of $P \cup \{v_1, v_2, v_3\}$ in its interior. Therefore, (p_1, p_2) is an edge in the augmented Delaunay triangulation of P.

The edge (p_1, p_2) partitions the interior of C_2 into two pieces, one that contains r and one that does not. It is possible to move the center of C_2 along the perpendicular bisector of (p_1, p_2) maintaining p_1 and p_2 on the boundary of C_2. There are two directions in which the center of C_2 can be moved to accomplish this. In one direction, say $\overrightarrow{d}$, the part of the interior that contains r only increases, so move the center in this direction until a third point $p_3 \in P \cup \{v_1, v_2, v_3\}$ is on the boundary of C_2. The resulting circle has the points p_1, p_2, and p_3 on its boundary and no points of P in its interior, so p_1, p_2 and p_3 are the vertices of a triangle t in the augmented Delaunay triangulation of P. The circumcircle $C(t)$ contains r in its interior and contains b either in its interior or on its boundary. In either case, t contradicts Statement 1, as promised. $\square$

Note that the first paragraph in the proof of Lemma 3 gives a method of testing whether $P = Q$, and when this is not the case, of finding a point in $Q \setminus P$. For each triangle t in the Delaunay triangulation of P, if t contains a blue vertex b then perform a pivot in R along the ray originating at b and passing through $C(t)$. If the result of this pivot is $C(t)$, then do nothing. Otherwise, the pivot finds a circle C with no red points in its interior and that has one blue point b and one red point $r \notin P$ on its boundary. By Lemma 2, the point r must be in Q. If t contains a red vertex, repeat the above procedure swapping the roles of red and blue. If both pivots (from the red point and the blue point) find the circle $C(t)$, then we have verified Statement 1 of Lemma 3 for the triangle t.

The above procedure performs at most two pivots for each triangle t in the augmented Delaunay triangulation of P. Therefore, this procedure performs $O(|P|) = O(\kappa)$ pivots. Since we repeat this procedure at most κ times before deciding that $\kappa < k$, we perform $O(\kappa^2)$ pivots, at a total cost of $O(\kappa^2 \times \frac{n}{\kappa^2} \log \kappa) = O(n \log \kappa)$. The only other work done by the algorithm is that of recomputing the augmented Delaunay triangulation of P each time we add a new vertex to P. Since each such computation takes $O(|P| \log |P|)$ time and $|P| \leq \kappa$, the total amount of work done in computing all these triangulations is $O(\kappa^2 \log \kappa)$.

In summary, we have an algorithm that given S and κ decides whether the condensed set Q of points in S that contribute to the decision boundary has size at most κ, and if so, computes Q. This algorithm runs in $O((\kappa^2 + n) \log \kappa)$ time. By trying increasingly large values of κ as described in Section 3.1 we obtain our main theorem.

Theorem 2. *The nearest-neighbour decision boundary of a set of n red and blue points in $\mathbb{R}^2$ can be computed in $O(n \log k)$ time, where k is the number of points that contribute to the decision boundary.*

Remark: In the pattern-recognition community pattern classification rules are often implemented as neural networks. In the terminology of neural networks, Theorem 2 states that it is possible, in $O(n \log k)$ time, to design a simple one-layer neural network that implements the nearest-neighbour decision rule and uses only k McCulloch-Pitts neurons (threshold logic units).

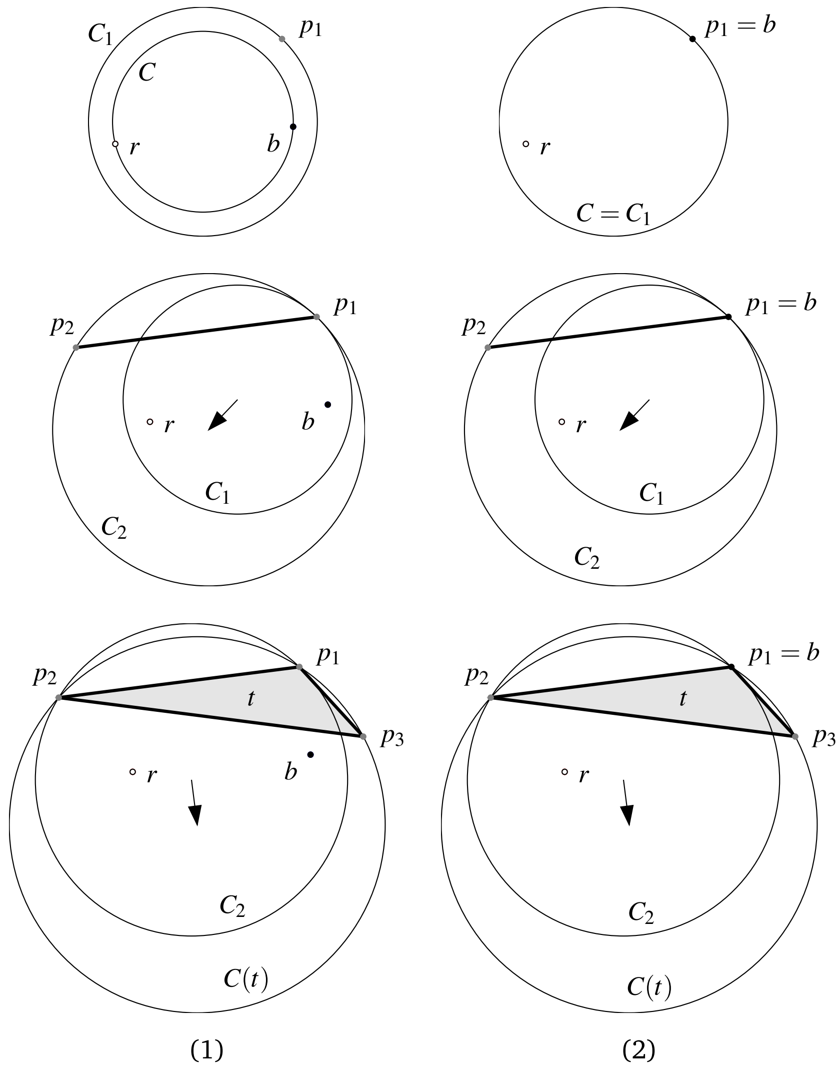


Figure 7: If $P \neq Q$ then Statement 1 of Lemma 3 is not true. The left column (1) corresponds to the case where $b \notin P$ and the right column (2) corresponds to the case where $b \in P$.

3.5 More than 2 Color Classes

Theorem 2 extends easily to the case where there are $c > 2$ colour classes and our goal is to find all Voronoĭ edges bounding two Voronoĭ cells of different colour. In this case we build a pivot data structure for each of the c colour classes. When performing pivots from a point in some colour class R, we perform queries in all the data structures except the one for colour class R. (This is equivalent to a pivot in the set $S \setminus R$.) This increases the cost of pivot operations to $O(\frac{cn}{m}\log m)$.

The only other modifications to the algorithm are the tuning of a few parameters. Note that c is a lower bound on k because each colour class contributes at least 1 point to the decision boundary. Therefore, we may assume that $c \le \kappa \le n^{1/3}$ since we can always start our algorithm with $\kappa = 2^{2^{\lceil \log\log c \rceil}}$ and we can use an $O(n \log n)$ time algorithm when $\kappa > n^{1/3}$. In the previous section, we chose $m = \kappa^2$, but could just as easily have chosen $m = \kappa^3$ without affecting the analysis of the running time (this is where the assumption that $\kappa \le n^{1/3}$ is used). With this choice of m, the time to perform a pivot becomes

$$O\left(\frac{cn}{m}\log m\right) = O\left(\frac{n}{\kappa^2}\log m\right) \quad .$$

Each round of the algorithm performs $O(\kappa^2)$ queries. Therefore, each round can be implemented in $O((\kappa^3+n)\log m) = O(n \log \kappa)$ time. The rest of the analysis remains unaffected and the entire algorithm runs in $O(n \log k)$ time.

Theorem 3. *The nearest-neighbour decision boundary of a set of n points from c different colour classes in $\mathbb{R}^2$ can be computed in $O(n \log k)$ time, where k is the number of points that contribute to the decision boundary.*

4 Conclusions

We have given $O(n \log k)$ time algorithms for computing nearest-neighbour decisions boundaries of bichromatic point sets in 1 and 2 dimensions, where k is the number of points that contribute to the decision boundary. A standard application of Ben-Or's lower-bound technique [1] shows that even the 1-dimensional algorithm is optimal in the algebraic decision tree model of computation. Simple variants of these algorithms give solutions for computing the nearest-neighbour decision boundary of points sets with $c > 2$ colours that run in the same time bounds.

We have not studied algorithms for dimensions $d \ge 3$. In this case, it is not even clear what the term "output-sensitive" means. Should k be the number of points that contribute to the decision boundary, or should k be the complexity of the decision boundary? In the first case, $k \le n$ for any dimension d, while in the second case, k could be as large as $\Omega(n^{\lceil d/2 \rceil})$. To the best of our knowledge, both are open problems.

Acknowledgements

This research wash initiated at the McGill Worshop on Instance-Based Learning at Bellairs Marine Biology Institute, Jan. 21–Feb. 6, 2003. The authors would like to thank the other workshop participants, namely, Greg Aloupis, Jit Bose, Vida Dujmović, Ferran Hurtado, Danny Krizanc, Henk Meijer, Mark Overmars, Tom Shermer, Sue Whitesides, and David Wood for helpful discussions and for providing a stimulating working environment. The authors are also grateful to an anonymous referee for showing us how to remove a $\log c$ factor from the running time in Theorem 3.

References

[1] M. Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

[2] B. K. Bhattacharya and S. Sen. On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *Journal of Algorithms*, 25(1):177–193, 1997.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computing and Systems Science*, 7:448–461, 1973.

[4] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

[5] T. M. Chan, J. Snoeyink, and C. K. Yap. Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional Voronoi diagrams. *Discrete & Computational Geometry*, 18:433–454, 1997.

[6] T. M. Cover and P. E. Hart. Nearest neighbour pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[7] B. Dasarathy and L. J. White. A characterization of nearest-neighbour rule decision surfaces and a new approach to generate them. *Pattern Recognition*, 10:41–46, 1978.

[8] L. Devroye. On the inequality of Cover and Hart. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:75–78, 1981.

[9] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of poyhedral intersection. *Theoretical Computer Science*, 27:241–253, 1983.

[10] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *Journal of Algorithms*, 6:381–392, 1985.

[11] C. A. R. Hoare. ACM Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[12] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[13] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.

[14] F. P Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[15] M. I. Shamos. Geometric complexity. In *Proceedings of the 7th ACM Symposium on the Theory of Computing (STOC 1975)*, pages 224–253, 1975.

[16] C. Stone. Consistent nonparametric regression. *Annals of Statistics*, 8:1348–1360, 1977.

[17] G. T. Toussaint. Proximity graphs for instance-based learning. Manuscript, 2003.

[18] G. T. Toussaint, B. K. Bhattacharya, and R. S. Poulsen. The application of Voronoi diagrams to non-parametric decision rules. In *Proceedings of Computer Science and Statistics: 16th Symposium of the Interface*, 1984.

[19] R. Wenger. Randomized quick hull. *Algorithmica*, 17:322–329, 1997.