Voronoi Diagrams and Nearest-Neighbor Search

Gabriel Ewing
Department of Electrical Engineering
and Computer Science
Case Western Reserve University
Cleveland, OH
gre5@case.edu

***Abstract*—We create Voronoi diagrams for two-dimensional maps with point obstacles. We explore the algebraic and geometric intricacies of the Sweepline algorithm used to build the diagrams. We review the concept of a k-Nearest-Neighbors classifier and use the completed Voronoi diagram to find nearest neighbors.**

I. Introduction

Robotic motion planning is the problem of finding a path from the starting configuration of a robot to some goal configuration that does not collide with any obstacles. In the case where the positions of the obstacles are known *a priori* but the start and goal configurations are not, it can be useful to create a *roadmap* of the configuration space. Roadmaps demonstrate connected edges through the free space such that a robot can easily move to some edge from a starting configuration, and easily move to the goal configuration from some edge. If there is a path between the two edges in free space, the roadmap should have a path between them.

A. Voronoi Diagrams

Voronoi diagrams are a class of roadmap. The basic definition of a Voronoi diagram is that its edges represent the points that are equidistant from the closest obstacles to that point. This means that each edge represents the path that maintains the maximal separation from two obstacles while traveling between them. Following the Voronoi edge between two obstacles minimizes the chance of a collision with an obstacle, which is a desirable property when the transition dynamics for the environment are non-deterministic and the cost of a collision is high. Visually, a Voronoi diagram divides the free space into *Voronoi regions* of points that share a common closest obstacle.

Voronoi diagrams are theoretically defined in spaces of arbitrary dimension and for obstacles of any shape. However, the complexity of creating the diagram grows quickly with the dimensionality of the configuration space. Additionally, more complex obstacle shapes bring more complex edges on the diagram. In this paper, we focus on two-dimensional environments with obstacles that are points.

B. Nearest-Neighbors Classification

Voronoi diagrams have other uses beyond robot motion planning. One application, which we explore in this paper, is to use a Voronoi diagram to build a *k-Nearest-Neighbors (KNN) classifier*. KNN classifiers come from the field of machine learning. The classification problem is as follows: given a *training set* of feature vectors with labels of either 0 or 1, produce a machine that, given a new feature vector *T*, will identify the *k* members of the training set with the closest feature vectors to *T*. The output label for *T* will be the class that the majority of the *k* nearest neighbors belong to. The metric used to define the "closest" feature vectors can vary based on the characterization of the available features, but the simplest way to measure it is to take the distance between the two points plotted in Euclidean space.

A complete Voronoi diagram can be used trivially as a KNN classifier with $k = 1$; finding the Voronoi region that a test sample belongs to is sufficient, as the nearest neighbor is the training sample associated with that region. For higher values of k, the classification problem is non-trivial. We explore how to implement such a system in this paper.

II. Methods

A. Creating Voronoi Diagrams

The problem of an efficient algorithm for creating Voronoi diagrams has been thoroughly explored in the literature. A naive method is to find the half-planes separating each obstacle and take the intersection of them to find a region. While this is relatively easy to implement, there are other algorithms that offer superior performance results [1]. One of the most prominent is the Sweepline algorithm introduced by Fortune [2]. Conceptually, the Sweepline algorithm moves a line (the eponymous Sweepline) through the plane containing the obstacles. In this paper, we will use the convention that the sweepline begins as a horizontal line at the obstacle with the highest y-value and moves downwards through the rest of the obstacles.

1) Incremental Construction: As we sweep down the plane, some obstacles will be above the sweepline and others will be below. The half-plane above the sweepline is then further separated into a region of points whose nearest obstacle

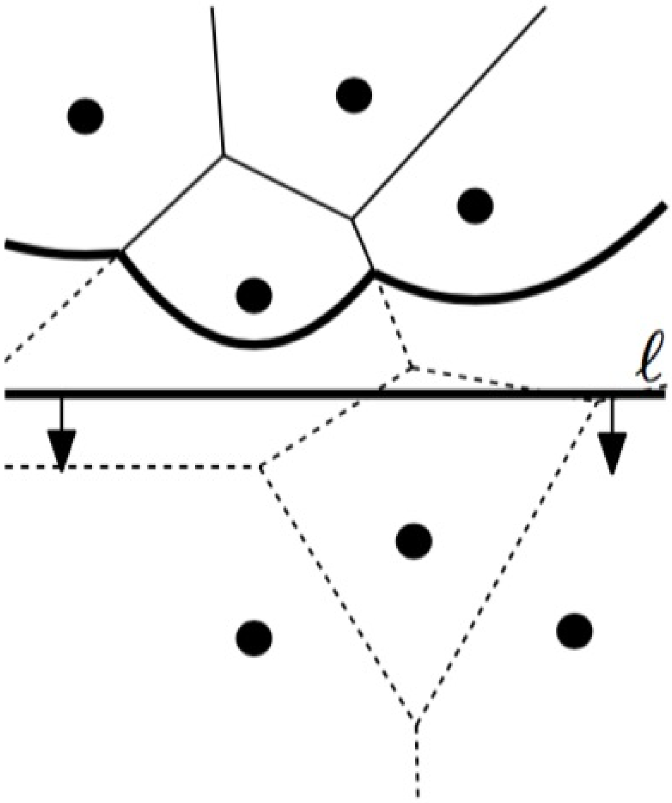

Fig. 1: Example of a beach line. The parabolic arcs are the beach line. The line segments above the beach line are finalized. The dashed line segments are the edges of the true Voronoi diagram that are not yet finalized. Taken from [3].

we can be sure of, and a region whose points may be closer to an obstacle below the sweepline than to any obstacle above the sweepline. The finalized and unknown regions are separated by a *beach line*.

The beach line consists of a set of parabolic arcs joined together at the intersections of the arcs. An example beach line is shown in Figure 1. Each arc of the beach line has a corresponding obstacle that is the closest point to it. Geometrically, the arcs take the form of parabolas because each obstacle is the focus and the sweepline is the directrix of a parabola. The points on the parabola are equidistant from the focus and the directrix.

The intersections of the beach line arcs are points that will be on Voronoi edges, equidistant from the obstacles corresponding to the two arcs meeting at that intersection. As the sweepline moves down, the arcs and the intersection do as well. A Voronoi edge is created by connecting the point where the intersection first occurred to the point where it disappears.

Computationally, it is impossible to move the sweepline down the plane in a continuous manner. We have to choose discrete points where the updates to the beach line will occur. Fortunately, there are a limited number of points that can change the structure of the beach line. We refer to these points as *obstacle events* and *circle events*.

2) Obstacle Events: Obstacle events occur when the sweepline encounters a new obstacle. That obstacle then must be inserted into the beachline. We locate the arc that the new obstacle is directly under and split it into two parts, one on either side of the new obstacle. At the moment when the obstacle is first encountered, it resides exactly on the beach line. This means that its "arc" in the beach line is actually given by a vertical line segment up to the existing arc that it splits. As the sweepline moves further down the plane, the arc widens and takes the usual parabolic curve.

In the special case where a new obstacle appears directly below an existing intersection in the beach line, we place the new arc between the two existing arcs and do not split either of them.

In general, two concave-up parabolas given by $y = Ax^2 + Bx + C$ can have a varying number of points of intersection. If the two have equal first derivatives and some vertical offset, there are no points of intersection. If the two have equal first derivatives and some horizontal offset, there is one point of intersection. If the first derivatives are unequal, there are two points of intersection. If the first derivatives are equal and there is no offset, there are infinite points of intersection as the parabolas are identical.

Fortunately, we only need to consider the cases with one and two points of intersection. The case with only a vertical offset is impossible during the sweepline algorithm, as a vertical offset means that the first derivatives will be different. The case with infinite points of intersection can be averted by eliminating repeated obstacles before diagram construction. The problem that remains is finding the one or two intersection points.

If the y-values are equal and there is only one intersection, the x-value of that intersection is the midpoint between the x-values of the minima of the parabolas. The equation for the x-value of intersections of parabolas with different y-values and foci a and b is given in [4] as

$$x = \frac{a_y b_x - \sqrt{a_y b_y((a_y - b_y)^2 + b_x^2)}}{a_y - b_y} \tag{1}$$

However, this only gives one intersection, when we know that there should be two. Working backwards through the derivation presented in [4], we find that the equation for both intersections is given by

$$x = \frac{a_y b_x \pm \sqrt{a_y b_y((a_y - b_y)^2 + b_x^2)}}{a_y - b_y} \tag{2}$$

Usefully, in the case where the focus with the lower y-value is on the sweepline, the two intersections given by this equation are equal. Additionally, given the x-values for the intersections, finding the y-values is simply a matter of plugging into the equations for either of the parabolas. Note that these equations are only valid after a translation where the y-value of the sweepline is subtracted from a_y and b_y, and $\min(a_x, b_x)$ is

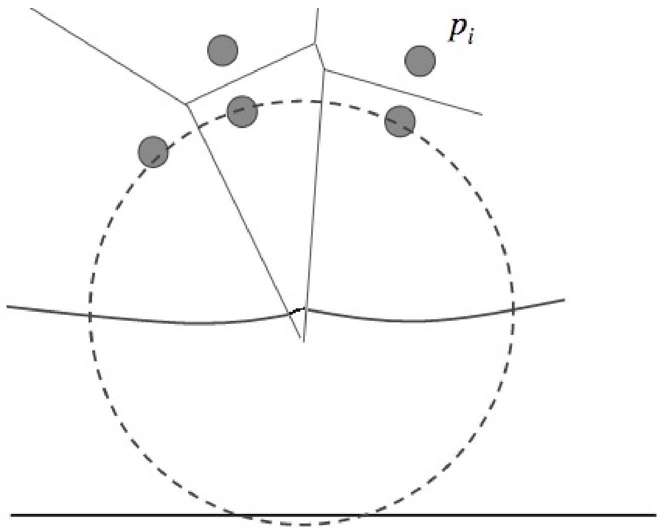

Fig. 2: Arcs converging to an intersection. The middle arc will soon be removed from the beach line. Taken from [1].

subtracted from a_x and b_x. The opposite translation has to be applied to the output intersections [4].

3) Circle Events: Circle events are when an arc is consumed by its neighbors and removed from the beach line. They are called circle events because the intersection of the Voronoi edges occurs at the center of the circle formed by the three points on the beach line, as shown in Figure 2. Furthermore, the event itself is the point on the circle with the lowest y-value; when the sweepline passes this point, there can be no other obstacles closer to the intersection.

However, if an obstacle is encountered within the circle before the circle event, the circle event is no longer relevant and is removed from consideration.

Not all sets of three consecutive obstacles have a circle event below them. The most common case for this is that with three such obstacles a, b, and c, ordered by increasing x-value, b lies below the line segment connecting a and c. This would mean that the circle event is above b, and therefore above the current position of the sweepline. Checking for this case amounts to computing the signed area of the triangle formed by a, b, and c, given in [4] as

$$A_{signed} = \frac{1}{2} \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} \tag{3}$$

If $A_{signed} < 0$, then the center of the circle is below b. Another case where the circle event does not need to be considered is that if a, b, and c are collinear, there is no circle containing all three of them.

B. Data Structures

There are two fundamental data structures used in the construction of the Voronoi diagram.

1) Beach Line: The first data structure represents the beach line. Optimally, this would be implemented as a balanced binary search tree such as an AVL tree or red-black tree. In this tree, leaf nodes represent obstacles and the internal nodes represent the intersections between arcs [1].

In the interest of ease of implemenation, we instead use a doubly-linked list to store the beach line. Nodes in the list contain obstacles, and two adjacent nodes have pointers to each other and to the growing Voronoi edge between them. Using a list instead of a search tree increases the complexity of locating the arc that a new obstacle falls under from $O(\log n)$ to $O(n)$.

In this setup, a node has two adjacent edges, each with an intersection at its endpoint. When a node is first encountered by the sweepline, its arc is a vertical line, and so only intersects the beach line once. However, the equation for intersections in Section II-A2 returns two copies of this intersection. This allows us to create the arcs without having to handle an additional case.

An obstacle can have multiple nodes associated with it. Each time the obstacle's parabola is split by an obstacle event, the single node containing that segment of the arc splits into two. This corresponds to the obstacle's Voronoi region gaining an incident edge.

When a circle event occurs, the corresponding obstacle's node is removed from the list and its neighbors are joined together, forming an intersection on the Voronoi diagram and a new edge leading out of it.

2) Event Queue: The second data structure is a priority queue containing the events, which we refer to as the *event queue*. We implement this using a binary heap, Python's *heapq*. Initially, the event queue contains all of the obstacles minus duplicate obstacles, beginning with the obstacle with maximal y-value. Ordering ties are broken by placing the event with the lower x-value first.

3) Interactions: There is some overlap between the responsibilities of the beach line and the event queue. Both of them keep track of circle events: each node in the beach line has an associated circle event, if any, and the event queue contains the active circle events in addition to the obstacle events. After each event, the nearby nodes in the beach line are examined for circle event updates. This includes both adding potential new circle events if a node was added to the beach line, and removing circle events if a new obstacle falls inside the circle. If a circle event is removed from the beach line, it also is removed from the event queue. This is a costly operation because lookup in a heap costs $O(n)$.

C. Nearest Neighbors

Given a complete and robust Voronoi diagram, KNN classification can be done in a similar manner to a breadth-first search. To classify a test point, the first step is to locate the Voronoi region that it resides in. The obstacle corresponding to that region is the nearest neighbor. If $k > 1$, the obstacles in regions adjacent to the nearest neighbor are added to a list. The second nearest neighbor will be the closest obstacle in the list. Additional neighbors can be added in a similar manner.

Classification in this manner requires that edges have knowledge of the obstacles on both sides of them, which our Voronoi implementation does not. We output a list of edges and a dictionary mapping each obstacle to the edges surrounding its region.

We modify the nearest-neighbor implementation as follows: after the nearest neighbor is found, it is removed from the list of obstacles and the Voronoi diagram is re-calculated. The second nearest neighbor is now the obstacle for the region that the test point resides in. This can be repeated for the k nearest neighbors. This impacts the computational complexity of the classification, as creating k Voronoi diagrams increases the runtime by a factor of k.

The standard Voronoi diagram has at its perimeter a number of edges that extend infinitely in the plane. It is difficult to efficiently calculate whether a point lies in one of these semi-bounded regions. Because of this, we take a post-processing step for the Voronoi diagram that imposes a bounding box around the obstacles; all the obstacles and intersections lie within the box, and any test points for classification must as well.

An additional post-processing step is to sort the Voronoi regions by the minimum value of y of any point in the region. This allows us to do a binary search to find the region with the highest y that is not entirely higher than the test point. From that point, we proceed linearly through the regions in order of decreasing highest y, checking whether the test point belongs in that region, until the correct region is found. This heuristic improves the runtime of the search for the region that the point falls into once we have the diagram, as all regions with higher minimum y can be ignored. The sorting step itself negates at least some of the runtime improvements, but in a better classification scheme (search within a diagram rather than re-calculating the diagram) it would likely be worth the one-time costs.

III. Results

A. Voronoi Diagrams

Figure 3 shows the results of creating a Voronoi diagram with fifty point obstacles with x and y coordinates randomly selected from a uniform distribution between 0 and 10. While we did not analytically verify the results, the output looks intuitively correct. Unfortunately, there is a latent bug in our implementation that causes an error with probability proportional to the number of obstacles, so this is close to the upper limit of the problem size that we can currently generate results for.

B. Nearest Neighbors

As a result of the aforementioned issues, testing the nearest-neighbors classification on any real dataset is not feasible; training generally requires significantly more than fifty input points. However, we are able to produce the k nearest neighbors to an input point, for values of k up to about 10. This means that the theoretical challenges are mostly handled.

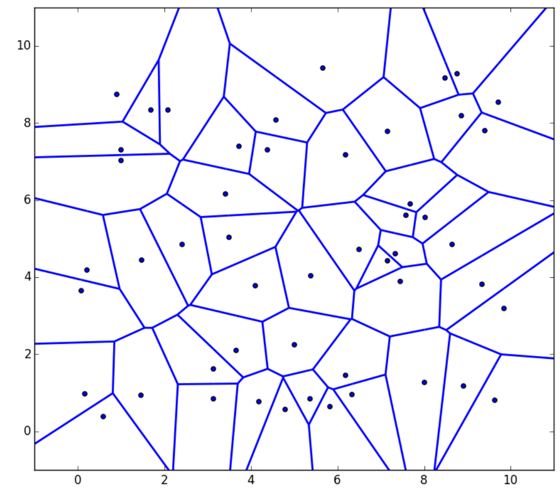

Fig. 3: Voronoi diagram for fifty randomly-generated points in a 10-by-10 box

Creating a more robust classifier would be mostly a matter of scaling and some amount of debugging. As mentioned in Section II-C, the algorithm that we used to find the neighbors was fairly inefficient, but on such a small dataset this did not impose any noticeable runtime degradation.

IV. Discussion

Our results are generally correct, but we wish that we could have implemented the most efficient algorithms and data structures for the diagram construction and KNN classification. During construction, we would prefer to have used a balanced binary tree rather than a doubly-linked list for the beach line. In the classification step, theoretical complexity improvements would have to be taken from modified data structures created during the diagram construction. If the edges list had included the vertices at either end of the edge, we could have made improvements in the time to find the region that a point belongs to. If the edges list had included the obstacles for the regions incident to the edge, we could have done an exhaustive search starting from the nearest neighbor rather than re-creating the Voronoi diagram. These improvements are theoretically simple to implement but we did not consider them during implementation of the Sweepline algorithm and lacked sufficient time to modify it at the end.

V. Conclusion

We successfully implemented a Voronoi diagram algorithm for point obstacles in two dimensions. While there exist easier Voronoi algorithms to implement, the Sweepline algorithm was interesting geometrically and promises good performance results. We used the completed Voronoi diagram to create a k-Nearest-Neighbors classifier that runs correctly on toy datasets.

References

[1] A. Miu, "Voronoi diagrams," 2001, http://nms.csail.mit.edu/~aklmiu/6.838/L7.pdf.

[2] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, no. 1-4, pp. 153–174, 1987.

[3] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*. Springer, 2000.

[4] K. Schaal, "HAVOC - here's another Voronoi code," 2013, diploma thesis. http://www.kmschaal.de/Diplomarbeit_KevinSchaal.pdf.