Dynamic Approximate Geodesic Nearest Neighbour Searching in Polygons with Holes

Lorenzo Theunissen

Supervisor:
Frank Staals

Second Supervisor:
Till Miltzow

July 14, 2023

Abstract

We present an algorithm and data structure to support approximate geodesic nearest neighbour queries for a set of sites S in a static polygon P with holes. Our data structure allows us to take a query point q from the set Q of query points that lie in P, and ask for the site in S closest to q. All distances are measured using the geodesic distance, that is, the length of the shortest path that is completely contained in P. We can construct this data structure in $O((n \log(n/\epsilon) \log n \log M)/\epsilon + (n \log M)/\epsilon^4 + (n \log^2 M)/(\epsilon^2 \log \log M) + (M \log^3 M)/\epsilon^2)$ time, using $O((M \log M)/\epsilon + (n \log n \log M)/\epsilon)$ space, and we can answer queries in $O((\log M)/\epsilon^4 + (\log^2 M)/(\epsilon^2 \log \log M) + (\log M \log n \log(n/\epsilon))/\epsilon)$ time, where M is the sum of the number of vertices, sites, and query points, that is $M = |P| + |S| + |Q|$.

Additionally, we present an algorithm and data structure for the simpler cases, sites in the plane, additive weighted sites in the plane, and sites inside a simple polygon. However, in these cases the query points can be arbitrary, and we can insert and delete sites from S.

In the plane, both weighted and unweighted, we show that we can create an $O((n \log^2 n)/\epsilon)$ space data structure in $O((n \log^2 n)/\epsilon)$ time, where we can answer queries in $O((\log^3 n)/\epsilon)$ time, and we can insert and delete sites in $O((\log^4 n)/\epsilon)$ time.

Inside a simple polygon, we show that we can create an $O((n \log^2 n + dn + m)/\epsilon)$ space data structure in $O(dn(\log m + \log n)/\epsilon + m \log m/\epsilon + n \log^2 n/\epsilon)$ time, where we can answer queries in $O(\log^4 n + \log^2 m)$ time, and we can insert and delete sites in $O((d(\log m + \log^2 n) + \log^3 n)/\epsilon)$ time. Where d is the depth of a tree we create, and this is at worst $O(m)$.

Keywords and phrases data structure, polygon with holes, geodesic distance, approximation algorithm, nearest neighbour searching, theoretical computer science, geometric algorithm.

Contents

Abstract **2**

1 Introduction **4**
1.1 Motivation 6
1.2 Research Questions 6
1.3 Naive solution 7

2 Related work **8**
2.1 Voronoi Diagrams 8
2.2 Shortest Path 11
2.2.1 Approximation Algorithms 12

3 Sites in the plane **13**
3.1 Unweighted Sites in a Plane 13
3.2 Weighted Sites in a Plane 15

4 Simple Polygon **17**
4.1 Finding the Closest Site for every Vertex 19
4.2 Putting it all together 20

5 Polygon with holes **21**
5.1 General Approach 21
5.2 Closest Sites on Separators 22
5.3 Temperature 24

6 Conclusion **27**
6.1 Future Work 27

1 Introduction

Nearest Neighbour searching is a classic problem in computational geometry. We are given a set S of sites, we want to preprocess the sites, such that for a query point q, we can find the nearest point $s \in S$. In our case, S is a dynamic set of points inside a polygon P which can contain holes. This project can be seen as an extension of the work from Agarwal, Arge, and Staals [3, 1], where a simpler variant of the same problem is investigated. In this variant they are investigating dynamic nearest neighbour searching in a simple polygon.

We are given a polygon P which can contain holes, and a set S of sites. When we query a point q inside the polygon, we want to be able to find the nearest site $s \in S$. The distance between two points p and q is the length of the geodesic $\Pi(p, q)$, that is the shortest obstacle avoiding path that lies completely inside the polygon, we call this the geodesic distance and it is denoted as $\pi(p, q)$. The Euclidean distance between two points p and q is denoted as $d(p, q)$. We can also add or remove sites from S, hence we have a dynamic nearest neighbour search. This case is the most general scenario, and we will call this the *fully dynamic* case, in this case the polygon and holes are static, but sites can be arbitrarily inserted and deleted, and we can query from an arbitrary point inside the polygon.

We also look at what we call the *offline*[1] variant. In this *offline* variant we have two different cases. We describe the general *offline* variant as follows:

We are given a polygon P which can contain holes, and two sets of points, a set of red points R, which are the possible query locations, and a set of blue points B, which are the sites. Each point has an associated temperature. When we query a red point, we want to find the closest blue site within a given temperature range, this range is the temperature of the query point plus or minus a threshold δ.

In the first *offline* case, we only use the sites, the set of blue points B, and the temperatures. In this variant we can do queries from arbitrary locations inside the polygon given a temperature range. We can use the temperatures of the sites in the construction of our data structure.

In the second *offline* case, we also have the set of red point R, and their temperatures. We can now use the knowledge of the query locations in the construction of our data structure.

When solving these problems we will use approximation algorithms. When we query a query point q, we will find the approximate closest site s. The geodesic distance between q and s will be at most $(1 + \epsilon)$ times the distance to the actual closest site to q. In the *offline* variants, the site s will still be within the temperature range. Only the distance will be approximated, whilst the temperature is still exact.

In section 3.1 we show how to solve the *fully dynamic* case in the plane. We can create an $O(n \log^2 n/\epsilon)$ space data structure in $O(n \log^2 n/\epsilon)$ time. We can answer $(1+\epsilon)$ approximate

[1] We use the word offline to describe the difference between fully dynamic, and knowing everything in advance. We are not using it in the traditional online vs offline scenario with a competitive score.[12]

closest site queries in $O(\log^3 n/\epsilon)$ time, and we can insert and delete sites from the data structure in $O(\log^4 n/\epsilon)$ time.

In section 3.2 we show how to solve the *fully dynamic* case in the plane where the sites have additive weights. We show that the additive weights don't influence the results, and that the unweighted case is simply a special case of the weighted case. In the weighted case, we can create an $O(n \log^2 n/\epsilon)$ space data structure in $O(n \log^2 n/\epsilon)$ time. We can answer $(1+\epsilon)$ approximate closest site queries in $O(\log^3 n/\epsilon)$ time, and we can insert and delete sites from the data structure in $O(\log^4 n/\epsilon)$ time.

In section 4 we show how to solve the *fully dynamic* case inside a simple polygon. We can create an $O((n \log^2 n + dn + m)/\epsilon)$ space data structure in $O(dn(\log m + \log n)/\epsilon + m \log m/\epsilon + n \log^2 n/\epsilon)$ time. We can answer $(1+\epsilon)$ approximate closest site queries in $O(\log^4 n + \log^2 m)$ time, and we can insert and delete sites from the data structure in $O((d(\log m + \log^2 n) + \log^3 n)/\epsilon)$ time. Where d is the depth of a tree we create, and this is at worst $O(m)$.

In section 5 we show how to solve the *offline* case inside a polygon with holes. We can create an $O((M \log M)/\epsilon + (n \log n \log M)/\epsilon)$ space data structure in $O((n \log(n/\epsilon) \log n \log M)/\epsilon + (n \log M)/\epsilon^4 + (n \log^2 M)/(\epsilon^2 \log \log M) + (M \log^3 M)/\epsilon^2)$ time, and we can answer $(1+\epsilon)$ approximate closest site queries in $O((\log M)/\epsilon^4 + (\log^2 M)/(\epsilon^2 \log \log M) + (\log M \log n \log(n/\epsilon))/\epsilon)$ time, where M is the sum of the number of vertices, sites, and query points.

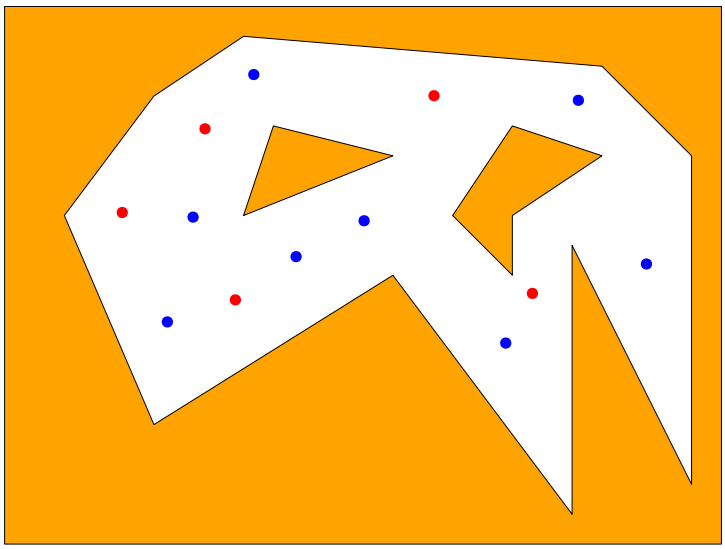


Figure 1: An example of a polygon with holes, a set of red points, and a set of blue points.

1.1 Motivation

The motivation for studying this problem originates in Ecology [8, 22]. The set of red points R, represents the locations at which animal or plant species lived in the past, and the set of blue points B, represents the locations where the species are today. Every point $p \in R \cup B$ has an associated environmental value which we call the temperature. The problem being solved, is to find for every species (red point), the closest location (blue point) it could have migrated to, whilst the temperature differs by at most δ. The environment is modelled by a polygon with holes, where the holes represent obstacles that the animals or plant cannot through or over, and therefore have to go around it. This can for example be a mountain or a lake.

1.2 Research Questions

Our goal with this study is to find an efficient algorithm for dynamic nearest neighbour searching inside a polygon with holes, where the distance is measured using the geodesic distance. We will make use of approximation algorithms to find efficient algorithms and data structures. To achieve this, we will use techniques from computational geometry to design algorithms and data structures, and we are met with the following research question.

1. Design an efficient algorithm and data structure such that for a red point, we can efficiently find the approximate nearest blue point with a similar temperature inside a polygon with holes.

Given the complexity of the problem, we will start with easier cases, and build up in complexity until we have reached the case mentioned above. We will also look at the more complex case, namely, the *fully dynamic* case in simpler environments.

2. Design an efficient algorithm and data structure such that for an arbitrary query point, we can efficiently find the approximate nearest (weighted) site in the plane, whilst efficiently allowing insertions and deletions of sites.

3. Design an efficient algorithm and data structure such that for an arbitrary query point, we can efficiently find the approximate nearest site inside a polygon, whilst efficiently allowing insertions and deletions of sites.

1.3 Naive solution

We can create a naive solution by making use of Voronoi diagrams. When given a set S of sites, a Voronoi diagram is the partition of the space into cells, such that every $s \in S$ is associated with a cell, and that for every point p inside a cell the closest site is s. A Voronoi diagram has a graph structure, the cells are the faces of the graph, the vertices of these cells are called Voronoi vertices, and the edges are called bisectors, on a bisector two sites are equidistant to it. On a Voronoi vertex, three or more sites are equidistant. A geodesic Voronoi diagram is also a partition of the space into cells, such that every $s \in S$ is associated with a cell, and that for every point p inside a cell the closest site is s, but the path to s is an obstacle avoiding path.

A naive solution for the problem can be constructed by using the shortest path maps from Hershberger and Suri [16] to create a Voronoi diagram inside a polygon with holes in $O((n+m)\log(n+m))$ time. We can store all the sites inside a balanced binary tree, sorted by their temperature. We can then have a Voronoi diagram for every node in the tree. This allows us to query the tree for the Voronoi diagrams we need, and then query the Voronoi diagrams to find the nearest site. The construction time of this tree is bounded by $O(nm \log n \log(n+m))$.

Proof. The tree has a height of $O(\log n)$, the first level of the tree (the root node) has a computation time of $O((n+m)\log(n+m))$, the second level has a construction time of $O((n+2m)\log(\frac{n}{2}+m))$, the third level has a construction time of $O((n+4m)\log(\frac{n}{4}+m))$, ..., the last level has a construction time of $O((n+\frac{nm}{2})\log(2+m))$. For every level we can state that the construction time is smaller than $O((n+nm)\log(n+m))$. With $\log n$ levels, we get a total construction time of $O(nm \log n \log(n+m))$. □

This solution is similar to, and is inspired by, the naive method mentioned by Arge and Staals [3] for a plane without obstacles.

2 Related work

In this section we will discuss work that is closely related or similar to the problem we are trying to solve. We will review Voronoi diagram algorithms, shortest path algorithms, and algorithms related to polylines.

2.1 Voronoi Diagrams

The paper by Arge and Staals [3] looks at the same problem, but in a simple polygon. This paper solves the same problem as we do, but instead of a polygon with holes, is solves it for a simple polygon. Given a set of sites, the algorithm gives a data structure of size $O(n \log m + m)$, it is a dynamic data structure, and sites can be inserted or deleted in $O(\sqrt{n} \log^3 m)$ time, and queries can be answered in $O(\sqrt{n} \log n \log^2 m)$ time. The algorithm recursively splits the polygon in half, and computes the geodesic Voronoi diagram for half of the polygon. This is done symmetrically for the other half. The geodesic Voronoi diagram for the sites in the left side of the polygon is only calculated in the right half of the polygon. This geodesic Voronoi diagram is a forest of bisectors. This forest can be stored as an Hamiltonian abstract Voronoi diagram, where only the degree one and degree three vertices need to be stored.

There is also an improved version by Agarwal, Arge, and Staals [1]. This paper is an improvement of the previous paper. Similar to the previous paper, the algorithm recursively splits the polygon in half. Instead of computing the geodesic Voronoi diagram of half the polygon, this improved algorithm computes a vertical shallow cutting of the geodesic distance functions. This leads to a data structure of size $O(n \log^3 n \log m + m)$, that can insert a site in $O(\log^5 n \log m + \log^4 n \log^3 m)$ amortised expected time, delete a site in $O(\log^7 n \log m + \log^6 n \log^3 m)$ amortised expected time, and answer queries in $O(\log^2 n \log^2 m)$ time. This data structure has an improved query time, whilst the space usage, insertion time, and deletion time have gotten worse, however, these times are still polylogarithmic.

For a Voronoi diagram inside a polygon we first have a simple algorithm by Aranov [4], Aranov gives an algorithm to calculate a geodesic Voronoi diagram for a polygon with m sides, and n point sites inside the polygon. The algorithm runs in $O((n+m) \log(n+m) \log m)$ time, and Aranov gives an algorithm that runs in $O((n+m) \log(n+m))$ time when the sites contain all the reflex vertices of the polygon. The algorithm uses a divide and conquer technique. The algorithm cuts the polygon P by a chord in two parts $P1$ and $P2$ of roughly the same number of edges. Then recursively the Voronoi diagram for P1 and P2 is calculated. The Voronoi diagrams of $P1$ and $P2$ need to be extended to get the full Voronoi diagram of P. The novel feature of the algorithm is the extension step. This step takes $O((n+m) \log(n+m))$ time, and is done by propagating from cut e through the triangles in preorder traversal. Two types of event can take place, a Voronoi cell dies out, or a vertex is reached, and it is assigned to the cell containing it.

A different and more efficient algorithm is given by Papadopoulou and Lee [23]. This paper gives an algorithm for a geodesic Voronoi diagram inside a polygon. The algorithm combines a sweep of the polygonal domain, and the merging step from a divide-and-conquer algorithm. This algorithm runs in $O((n+m)\log(n+m))$ time. This paper also introduces the concept of subcells, where a cell is a Voronoi region, a subcell is a maximum set of points that have the same anchor. They also introduce the concept of a breakpoint, this is an endpoint of a bisector that is not a Voronoi vertex. This is an intersection between a Voronoi edge, and an edge of the shortest path map. The Voronoi diagram can be seen as an embedding of a planar graph. The polygon gets triangulated, and the dual graph of the triangulated polygon is a tree, and an arbitrary node of this tree is assigned as the root. Using a sweep algorithm, going over the triangulated polygon in postorder traversal of the rooted tree, a Voronoi diagram inside each triangle is calculated for the sites below. A second sweep is done in preorder traversal, to get the Voronoi diagram inside each triangle for the sites above. In the third step the two subdivisions are merged, to get a complete Voronoi diagram.

This algorithm was further improved by Liu [19]. This paper builds on the ideas of the previous paper. It uses the same approach of triangulating the polygon, doing two sweeps over it, to get partial Voronoi diagrams, and merging these two, to get a complete Voronoi diagram. This paper uses a wavefront algorithm to calculate the incomplete Voronoi diagrams. This allows for the construction of the Voronoi diagram in $O(n+m(\log m+\log^2 n))$ time, which is optimal for $m = O(\frac{n}{\log^2 n})$, and the $\log^2 n$ terms comes from the computation time of calculating a single Voronoi vertex.

A paper that looks more broadly at different types of Voronoi diagrams is by Oh and Ahn [21]. This paper looks at the geodesic nearest-point, higher-order, and farthest-point Voronoi diagrams with a set of S of m points inside the n-gon. They show an $O(n+m\log m\log^2 n)$-time algorithm for nearest-point, an $O(k^2 m\log m\log^2 n + min(nk, n(m-k)))$-time for an order-$k$ Voronoi diagram, and an $O(n+m\log m+m\log^2 n)$-time algorithm for an farthest-point Voronoi diagram. The nearest-point algorithm is optimal for $m \leq n/\log^3 n$. All algorithms are based on a polygon sweep. A nearest-point Voronoi diagram in a plane can be calculated by sweeping the plane, during the sweep the partial Voronoi diagram of the sites above the sweep line is computed, which becomes a complete Voronoi diagram by the end of the sweep. This is done by defining two types of events. The nearest-point algorithm fixes a point o on the boundary, and they move a point x from o in a clockwise direction, and the point x is used as a sweep point. They first compute a topostructure of the Voronoi diagram, that consists of the degree-1 and degree-3 vertices, and the adjacency graph of the Voronoi cells, from this structure the complete Voronoi diagram is calculated. Inside the sub-polygon induced by o, x, and the polygon boundary, they define a region for every site s inside the sub-polygon $R_s(x)$, that is every point p in the sub-polygon for which $d(p,s) \leq d(p,\pi(o,x))$ holds. They call the union of all the regions $R_s(x)$, $R(x)$. To compute the Voronoi diagram restricted to $R(x)$, you only need sites inside the sub-polygon. After the sweep, $R(x)$ might not coincide with the polygon, so solve this, they add a very thin triangle to the boundary, to make a bit larger polygon, this solves the issue, and the complete Voronoi diagram can

be calculated.

There is also the paper by Kunze, Wolter, and Rausch [17], which looks at geodesic Voronoi diagrams on parametric surfaces. This paper uses a divide and conquer strategy to calculate the Voronoi diagram of a parameterised surface. The distance between two points on the surface is the path with the shortest length on the surface, and is called the geodesic distance. They define $M(p, q)$ of two points p and q, to be the equidistance set, this set divides the surface into two sub regions. On a Euclidian surface, $M(p, q)$ is easy to calculate and is a straight line perpendicular to the line segment that connects p and q. To calculate $M(p, q)$ the authors use the first and second order differential equations of the surface. From these equations, a geodesic circle of a point p can be calculated. This can be used to calculate $M(p, q)$ of two points. The algorithm recursively splits the set of sites into 2, and if there are only 2 sites, it calculates $M(p, q)$, and recursively merges the subdiagrams, this results in the complete Voronoi diagram.

In addition to Voronoi diagrams, there are also approximate Voronoi diagrams, when querying in a $(1 + \epsilon)$-approximate Voronoi diagram, the returned site is at most a factor of $(1 + \epsilon)$ further away than the true closest site. The papers by Arya, and Malamatos [6], and the extension by Arya, Mount, Netanyahu, Silverman, and Wu [7]. This paper gives an approximate Voronoi diagram in $\mathbb{R}^d$, it gives an (t, ϵ) approximation. The cells of the Voronoi diagram have a constant complexity, and every cell is associated with t representative points of S, such that for any point in a cell, the nearest representative is within a factor of $(1 + \epsilon)$ of the nearest site. This paper uses the concept of a well-separated pair decomposition. If S is a set of n points, two subsets X and Y are well-separated if they can be enclosed by two disjoint balls of radius r, such that the distance between the centres of the balls is at least αr, where α is the separation factor. They also introduce a balanced box-decomposition, this is a unit hypercube in $\mathbb{R}^d$, with a recursive quadtree. They encompass the space with a balanced box-decomposition, bu scaling and translating the space, so everything is inside a unit hypercube. By computing a well-separated pair decomposition for all pairs, they can use the information of the balls, to calculate a set of quadtree boxes that are used, and are stored in a tree. The subdivision acquired in the leaves, is an approximate Voronoi diagram.

Finally, we have a paper by Aronov, de Berg, and Thite [5]. This paper gives tight bounds on the complexity of the bisectors of a Voronoi diagram on a realistic terrain. In this paper a terrain is a polyhedral terrain, a triangulated mesh where each vertex has an elevation. On a terrain the complexity of the bisectors can get high, $\Omega((n + m)m)$, where m is the number of triangles of the terrain. However, this happens on carefully constructed terrain. The paper defines a set of 4 properties, and if all 4 hold, a terrain is realistic. The number of breakpoints is at most the number of triangles. The complexity of a bisector is defined as its number of break points plus the number of it intersects a terrain edge. The complexity is $O((\epsilon + \lambda)n)$, where ϵ is the slope of the terrain and λ is its density. The density of a set S of objects the smallest number λ such that any disk D intersects at most λ objects $o \in S$ such that the diameter of $o \leq$ than the diameter of D. Our set of objects is the set of edges of the triangulation of the terrain. The complexity of a Voronoi diagram it proportional to

$(m + n)$ plus the number of times a Voronoi edge intersects a terrain edge. It can then be shown that the total complexity of a Voronoi diagram on a realistic terrain is in the worst case $\Theta(n + m * \sqrt{n})$.

2.2 Shortest Path

We have looked at shortest path algorithms, because Voronoi diagrams, and shortest path searches are closely related. Our goal is to find the nearest neighbour within a temperature range, and a Voronoi diagrams does not have to be the solution. One of the papers is by Mitchell [20]. This paper gives a shortest path algorithm that avoids obstacles in the plane. Given a point s, it can give the shortest path to any point p in the plane. It runs in $O(kn \log^2 n)$ where n is the number of vertices, and k is the illumination depth of the obstacles, and it uses $O(n)$ space. The algorithm computes a shortest path map. First the visibility polygon of point s is calculated. A set of seen obstacles is defined, and that is the current shortest path map. A point is accessible if it can be reached by the shortest path map without bending at an unseen obstacle vertex, Vertices on the boundary of the region of accessibility, and were unseen, are now seen. This process, of computing accessibility, and updating the shortest path map, is iterated until there are no unseen vertices left. k is the number of iterations of this algorithm.

A different algorithm comes from Hershberger and Suri [16]. This paper gives an optimal algorithm to find the shortest Euclidean path in the plane. They use a technique called continuous Dijkstra, this technique simulates a wavefront from a source in the presence of obstacles. The boundary of the wavefronts is made of circular arcs, called wavelets. An arc is generated by the vertex of an obstacle that is already covered by the wavefront. Simulating the wavefront is done using events, there are two types of events, wavefront-wavefront collisions, and wavefront-obstacle collisions. To do this efficiently, the paper subdivides the plane into a quad-tree-style structure of size $O(n)$, and it uses an approximate wavefront. Each cell in the subdivision, called a conforming subdivision, has useful properties. The most important one being, for any edge e of the subdivision, there are $O(1)$ cells within distance $2|e|$ of e. After the subdivision, the obstacle edges are added back in, whilst maintaining the conforming property, and that leads to there being $O(1)$ cells withing shortest path $2|e|$ of e, for a non-obstacle edge. These cells are used to propagate the wavefronts. Wavefront-obstacle events are easy to handle. There are two types of wavefront-wavefront events, the collisions between two non-neighbouring wavelets are hard to detect. Instead they use an approximate wavefront, instead of computing the wavefront exactly, it is separates into two different wavefronts, coming front opposite sides of the edge. These wavefronts approximate the wavefront hitting the edge from one side. They use timers to estimate when an edge is hit with a wavefront, and discard any part of the wavefront that arrives after a timer at that edge has gone off. After the propagation phase, they use Voronoi diagram techniques inside each cell to compute the precise collision events in that cell, those collisions determine the final shortest path map.

2.2.1 Approximation Algorithms

We also looked at approximate nearest neighbour and shortest path algorithms. The paper by Poudel [24] looks at the shortest path in a graph. This paper gives an algorithm to compute an approximate shortest path in a graph by using a distance oracle, and runs in $(CS^{*O(dim)})$, where C is a constant, S^* is the number of vertices in the shortest path, and dim is the doubling dimension of the graph. The goal of this paper was to get an algorithm that prunes more nodes than traditional shortest path algorithms. This is done by only expanding nodes that are on a $(1+\epsilon)$ shortest path. The distance oracle gives an approximate shortest distance between two nodes, this is used to determine if a node is on a $(1+\epsilon)$ shortest path, and the node will only be expanded if it is.

The paper by Thorup [25] gives approximate distances. This paper gives an algorithm to compute an approximate distance between points in a plane with obstacles, where n is the number of corners/vertices. They pick an arbitrary corner, and they compute an exact shortest path tree, that includes all the corners. With this tree, and the edges of the obstacles, the plane is triangulated. This triangulation is a planar graph where the shortest path tree is the spanning tree. You can find a triangle, and if you look at the region of the fundamental cycles induced by any side of the triangle, then this cycle will have less than half of the triangles. A union of at most 6 shortest paths is a separator. This are the at most 3 paths from the triangle corners in the triangulation, plus any triangle side which is not inside an obstacle, or in the exterior region. For a given shortest path Q, it can be shown that the shortest path between u and v, going through Q, has at most $O(4k)$ connections. With connections from u to Q, and v to Q, the distance between u and v can be estimated. This is done recursively by splitting the plane along the 6 shortest paths, and recursing in the induced regions. Connections can now be found recursively and have a stretch of at most $\epsilon = \frac{1}{k}$.

The paper by Li and Dunson [18] approximates a geodesic distance over a surface. This paper gives a distance approximation for a given set of points following a geometric shape, instead of the Euclidean distance. They assume that M is a smooth compact Riemannian manifold. First they calculate a local estimation, and use that to get a global estimation for arbitrary points. The local estimation is calculated using a spherical distance, spheres approximate the manifold better than lines. They give a method to compute the spheres that are needed, and how this is implemented for different manifolds.

3 Sites in the plane

In this section, we will provide the foundation to solve the case in a simple polygon. We first prove that we can do approximate nearest neighbour searches in the plane. We will then show that we can do approximate nearest neighbour searches with additively weighted sites in the plane.

3.1 Unweighted Sites in a Plane

In this section we will show that given a set of sites in the plane, we can create an $O((n \log^2 n)/\epsilon)$ space data structure in $O((n \log^2 n)/\epsilon)$ time, with an $O((\log^4 n)/\epsilon)$ insertion and deletion time, and we can answer queries in $O((\log^3 n)/\epsilon)$ time.

We have a set S of sites and a cone c, every site $s \in S$ is inside c. c has an origin c_o and a direction c_d. This direction is the bisector of the two lines of the cone. We consider the half-line l that starts at c_o and goes in the direction of c_d. We take every site in S, and project it on l, we call this set S'. If we have a query point q, we can look around q in k evenly spaced cones. In each cone we find the approximate closest site, and we return the overall closest site. We will use $\frac{1}{k}$ as our ϵ value for our $(1+\epsilon)$ approximation.

Lemma 1. *Given that s is the closest site to q in S, and s' is the closest site to q in S'. s' has a distance to q of $(1+\epsilon)d(s,q)$, that is $d(s',q) \leq (1+\epsilon)d(s,q)$.*

Proof. Without loss of generality, assume that our cone is facing to the right, that is the direction of the cone can be represented by the 2D vector (1,0), see Figure 2. We claim that the left most site inside the cone, is approximately the closest site in the cone, if we project every site in S to S', the closest site to q is the left most site. Consider two points inside this cone, points a and b. The points a and b have the same position in S' but not in S. If we maximise the difference between $d(q,a)$ and $d(q,b)$, we get that $d(q,a) \leq (1+\epsilon)d(q,b)$.

Looking at Figure 2 we have site a, and site b, site a can be considered the closest site, whilst site b is actually closer. The difference between the distances $d(q,a)$ and $d(q,b)$ is the largest when a lies on the cone, and b lies on the bisector of the cone. The angle between $d(q,a)$ and $d(q,b)$ is $\frac{2\pi}{2k} = \frac{\pi}{k}$ With some simple trigonometry, we find that $d(q,a) = \frac{1}{\cos(\frac{\pi}{k})}x$, we want that $\frac{1}{\cos(\frac{\pi}{k})} \leq 1 + \frac{1}{k}$, and this is the case for $k \in \mathbf{N}^+, k \geq 6$. □

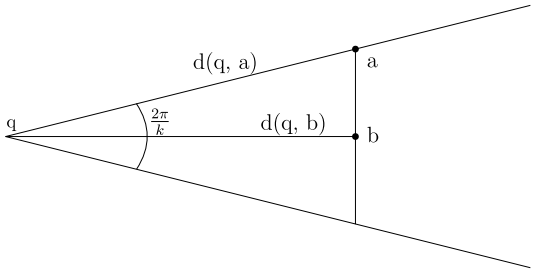


Figure 2: An example of one of the k cones pointing to the right.

With Lemma 1 we can give a data structure and an algorithm to solve the *fully dynamic* version of the problem for sites in the plane. To find the closest site to a query point q, we will use a three dimensional range tree from de Berg, Cheong, van Kreveld, and Overmars [11] for every cone direction. When we look at a cone, we can define three properties, the direction of the cone, and the two lines that make up the cone, l_1, and l_2. These lines have a starting point c_o, and go infinitely in one direction. We consider the lines with an extension in the other directions, l_1' and l_2'. We consider these lines as axes, meaning that a point will not be stored in terms of x and y position, but in terms of l_1' and l_2'. In a regular range tree, the points get sorted by their x and y position. We project the points on the lines l_1' and l_2', and sort them according to this projection. This sorts the points in the direction of these lines. We use these two axes and the direction of the cone, as our three dimensions for our three dimensional range tree.

A single cone has a construction time of $O(n \log^2 n)$, a space usage of $O(n \log^2 n)$, a query time of $O(\log^3 n)$, and an insertion/deletion time of $O(\log^3 n)$. With k cones the construction time is $O((n \log^2 n)/\epsilon)$, the space usage is $O((n \log^2 n)/\epsilon)$, and the query time is $O((\log^3 n)/\epsilon)$. The range trees from de Berg, Cheong, van Kreveld, and Overmars [11] don't mention insertions and deletions, meaning that if we do insertions and deletions, the trees will not remain balanced. However, we can use techniques from Anderson [2]. This gives us that the cost of rebalancing a subtree v is $O(P(|v|))$, and the amortised cost of an update is $O((P(n)/n) \log n)$. This allows for an amortised insertion and deletion time of $O(\log^2 n)$ in a one dimensional tree. In our three dimensional tree, this allows for amortised insertions and deletions in $O(\log^4 n)$. In general this adds an $O(\log n)$ factor to all the insertions and deletions in a tree, and we will use this in all our tree structures going forward.

This results in the points being sorted according to l_1' in the first level of the tree, according to l_2' in the second level, and finally according to the cone direction in the final level. When we query a point q in this tree, the first two levels give use the points that lie inside the cone starting at q, and the third level returns the closest site in the direction of the cone. By Lemma 1, this returns a site that has a distance to q with a $(1 + \epsilon)$ approximation of the closest site.

Theorem 2. *Given a set S of sites, we can create an $O((n \log^2 n)/\epsilon)$ space data structure in $O((n \log^2 n)/\epsilon)$ time. With this data structure we can find a $(1+\epsilon)$ approximate closest site to an arbitrary query point in the plane in $O((\log^3 n)/\epsilon)$ time, and we can insert and delete sites from the data structure in $O((\log^4 n)/\epsilon)$ time.*

3.2 Weighted Sites in a Plane

In this section we now consider the scenario in which every site also has an additively positive weight, we now no longer use the Euclidean distance to find the closest site, we consider the distance to be the Euclidean distance plus the weight of the site. We will show that given a set of additively weighted sites in the plane, we can create an $O((n \log^2 n)/\epsilon)$ space data structure in $O((n \log^2 n)/\epsilon)$ time, with an $O((\log^4 n)/\epsilon)$ insertion and deletion time, and we can answer queries in $O((\log^3 n)/\epsilon)$ time.

Lemma 3. *Given that s is the closest site to q in S, and s' is the closest site to q in S'. s' has a distance to q of $(1+\epsilon)(d(s,q)+w_s)$, that is $d(s',q)+w_{s'} \leq (1+\epsilon)(d(s,q)+w_s)$.*

Proof. We will use a slightly modified version of the approach we used for the unweighted version. When we compare the sites in the direction of the cone, we add the weight to the site as a unit vector in the cone direction, see Figure 3. This way, we essentially move all sites in the cone direction by their weight, because the weights are additive, and the distance is measured in the cone direction, moving the sites to the right and removing their weights, is the same as not moving them and adding their weight. This brings us back in the unweighted variant, and we can use Lemma 1 to prove that the weighted variant is correct. □

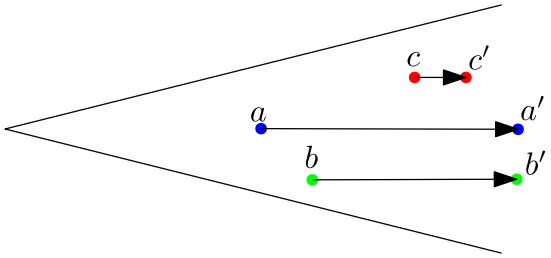


Figure 3: Moving sites, sites a, b, and c move to the right by their weights.

With Lemma 3 we can give a data structure and an algorithm to solve the *fully dynamic* version of the problem for sites with additive weights in the plane. We will use a slightly modified version of the approach we used for the unweighted version. When we sort in the direction of the cone, at the third level of our three dimensional range tree, we add the weight to the site as a unit vector in the cone direction. This way, we essentially move all sites in the cone direction by their weight, see Figure 3. When making the trees for the lines of the cone, l'_1 and l'_2, we use the original locations of the sites. Therefore, when we do a query, we only report the sites that are actually inside the cone without moving the sites,

but we take the weight into account when querying the closest site inside the cone. This modification does not add to our construction time nor our query time, so we still have an $O((n \log^2 n)/\epsilon)$ construction time, a space usage of $O((n \log^2 n)/\epsilon)$, an $O((\log^3 n)/\epsilon)$ query time, and an $O((\log^4 n)/\epsilon)$ insertion/deletion time.

Theorem 4. *Given a set S of sites, we can create an $O((n \log^2 n)/\epsilon)$ space data structure in $O((n \log^2 n)/\epsilon)$ time. With this data structure we can find a $(1+\epsilon)$ approximate closest additively weighted site to an arbitrary query point in the plane in $O((\log^3 n)/\epsilon)$ time, and we can insert and delete sites from the data structure in $O((\log^4 n)/\epsilon)$ time.*

4 Simple Polygon

In this section we look at the case with a simple polygon, where all the sites will be inside the polygon. n will be the number of sites, and m will be the number of vertices of the polygon. We will show that given a set of sites inside a simple polygon, we can create an $O((n \log^2 n + dn + m)/\epsilon)$ space data structure in $O(dn(\log m + \log n)/\epsilon + m \log m/\epsilon + n \log^2 n/\epsilon)$ time, with an $O((d(\log m + \log^2 n) + \log^3 n)/\epsilon)$ insertion and deletion time, and we can answer queries in $O(\log^4 n + \log^2 m)$ time. d is the depth of a tree structure that we create, and is at worst $O(m)$.

We will handle sites and vertices separately. We will use sites as unweighted points, and we will use vertices as weighted points. The weight of a vertex v is the geodesic distance to the closest site s, and v knows s. We will explain the process for a single cone direction, the process is analogous for all cone directions, and we can get the closest overall site by querying all cones.

The global idea is to use the observation that the closest site either has a direct path to the query cone, or the path has to go through a vertex. If every vertex knows the closest site, and uses the distance to this site as a weight. We can use Lemma 1 and 3 to find the closest site by only looking at sites that have a direct path, and vertices that have a direct path.

We preprocess the polygon using a trapezoidal map from de Berg, Cheong, van Kreveld, and Overmars [11, chapter 6] in the cone direction, we then query all the sites in the trapezoidal map in the cone direction, for every site $s \in S$ we find the first edge intersection i, we record the distance to it, and save a line segment from s to i. This structure allows for $O(\log m)$ expected query time, can be constructed in $O(m \log m)$ time, and uses $O(m)$ expected space.

After we have done this for all the sites, we have $O(n)$ parallel line segments, we will save these in a tree using the data structure from de Berg, Cheong, van Kreveld, and Overmars [11, chapter 10]. In this data structure the lines in the tree are parallel, and the query line segment is perpendicular. We always query from the same angle, and this angle is not parallel to the line segments, therefore, we can apply a shear to make the query segments perpendicular.

We do the same for the vertices and this gives us a tree for the $O(m)$ vertex segments. We will make these maps for all k directions, and for the directions of the cone lines. When we have a query cone, we can find where it intersects the polygon, and we can then use the cone as two line segments, see Figure 4.

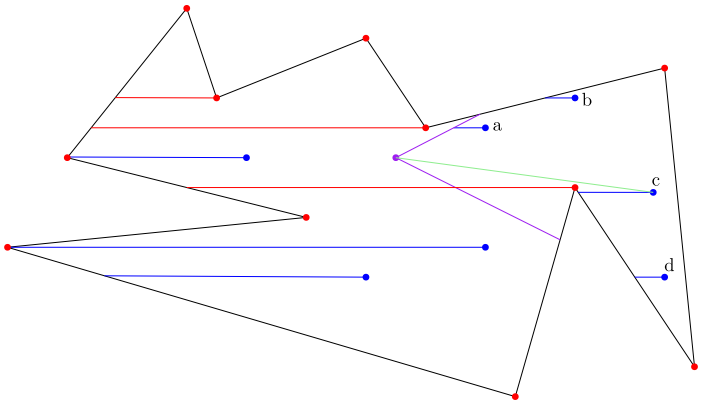


Figure 4: A simple polygon with a right facing cone, and the line segments of the sites and vertices from the opposite direction. In the area of the cone are 4 sites, one is *unblocked*(a), one hits the same edge of the polygon as the cone (b), one has line of sight to the query point, but is considered *blocked*(c), and one is completely *blocked*(d).

We will only consider the sites and vertices that when you draw a line to the left, intersect the cone before they intersect an obstacle, and sites that intersect the same edge as the cone, see site b in Figure 4. If we do this, we can discard sites that have a distance closer to an obstacle in the opposite cone direction, and only consider *unblocked* sites. We consider a site *unblocked* if and only if the line drawn to the left intersects the cone boundary. In Figure 4 we have 4 sites inside the cone area, of which one is *unblocked*, one intersects the same edge as the cone, and two are hidden.

Let s be the approximate closest site, s is either *unblocked* or *blocked*. If s is *unblocked* we will find s as the closest site by the use of Lemma 1 and 3. If s is *blocked*, s has to go through a vertex v that is *unblocked* in the cone. v has the distance to s as its weight, and will be found as the closest point by the use of Lemma 3.

During the construction of the data structure, we know for every site which edge they intersect, and the distance form the site to the edge. On edges with intersections we store a two dimensional binary tree, in the first dimension of the tree we store the positions of the intersections sorted from one side of the edge to the other, in the case of a right facing cone, this is sorted from left to right alongside the edge. In the second dimension of the tree we store the weighted distance to the site, this weighted distance is the sum of how far along the edge it is, and the actual distance to the site. During a query, if we hit an edge, we can find the closest site on this edge in $O(\log^2 n)$ time, the first dimension of the tree returns all the sites to the right of the intersection point, and the second dimension of the tree returns the closest weighted site. We only insert every site k times, giving us a construction time of $O(n \log^2 n/\epsilon)$, and a space usage of $O(n \log n/\epsilon)$. We can insert or delete a site in $O(\log^3 n/\epsilon)$ time.

4.1 Finding the Closest Site for every Vertex

We preprocess the polygon so we can answer two-point shortest path queries in $O(\log m)$ time using a data strucure from Guibas and Hershberger [13](see also the follow up note of Hershberger [15]). We can find the the closest site for every vertex by creating a data structure based on the trapezoidal maps we made in section 4. We will again consider the case for one cone direction, it is analogous for the other directions. If we have a query cone to the right, we are concerned about hidden areas, i.e, areas that if they contain a site, the site does not have a line segment to the cone.

We can see that such an area is a subpolygon that is created by drawing a line to the right from a vertex, we can also see that this subpolygon can be seen as the union of trapezoids in the trapezoid map. This means that if this vertex is inside a query cone, the closest site we need to consider for this vertex lies within this subpolygon. If we insert a site into the polygon, we can query the trapezoid map to find in which trapezoid the site lies, and therefore in which subpolygon the site lies. When inserting a site in this manner, we know to which vertex it belongs, in $O(\log m)$ time. For this vertex we store a sorted list of all the sites that are inside its subpolygon, and we sort the sites by their distance to the vertex, which can be calculated in $O(\log m)$ time each. If we look at Figure 5, we can see that these subpolygons can nest.

Due to how the subpolygons are created, every subpolygon is either a subpolygon of the entire polygon or a subpolygon of another subpolygon. using these relations, we can store these subpolygons in a tree structure. When we insert a site, we insert it into the lists of all the vertices form which it is inside a subpolygon. We call the depth of this tree d, this depth can be $O(m)$ in the worst case, but is determined by the nature of the polygon. When a vertex is found in a cone, we can find the nearest site in $O(\log n)$ time. To store the lists of sites that the vertices hold, we need $O(dn)$ storage, we only need to store sites multiple times if they are in multiple subpolygons, and therefore the storage is $O(dn)$ and not $O(mn)$.

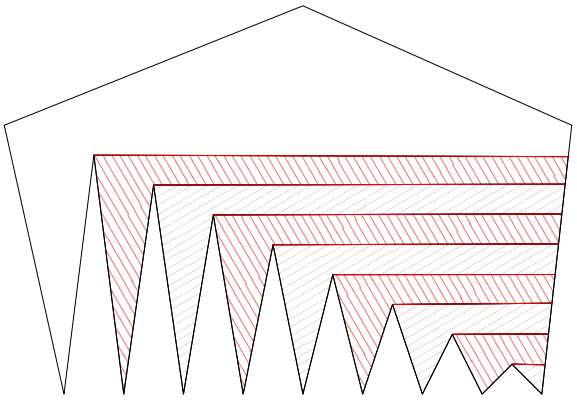


Figure 5: A simple polygon with nesting subpolygons.

4.2 Putting it all together

The total construction time is the sum of the time to construct the trapezoidal maps, $O((m \log m)/\epsilon)$ time, the time to construct the segment trees, $O((m \log m)/\epsilon)$ time, and the time to insert the sites one at a time, $O(dn(\log m + \log n)/\epsilon)$. The total construction time is $O(dn(\log m + \log n)/\epsilon + (m \log m)/\epsilon)$.

Using the segment tree data structures, we can query the closest *unblocked* site, and the closest weighted *unblocked* vertex. Doing this in all k cone directions, gives use the approximate closest site. Querying the segment trees takes $O(\log n + \log m + z)$ time, where z is the amount of intersections with the cone. Instead of returning all the intersections, we can get the results back in a tree structure, where all the points are sorted in the cone direction. The vertices have additive weights, where the weights are the distances to the nearest site. This takes the query time down to $O(\log^2(n) + \log^2(m))$, and this gives us a total query time of $O(\log^4(n) + \log^2(m))$. We get a total construction time of $O(dn(\log m + \log n)/\epsilon + m \log m/\epsilon + n \log^2 n/\epsilon)$, and we can do an insertion/deletion in $O((d(\log m + \log^2 n) + \log^3 n)/\epsilon)$.

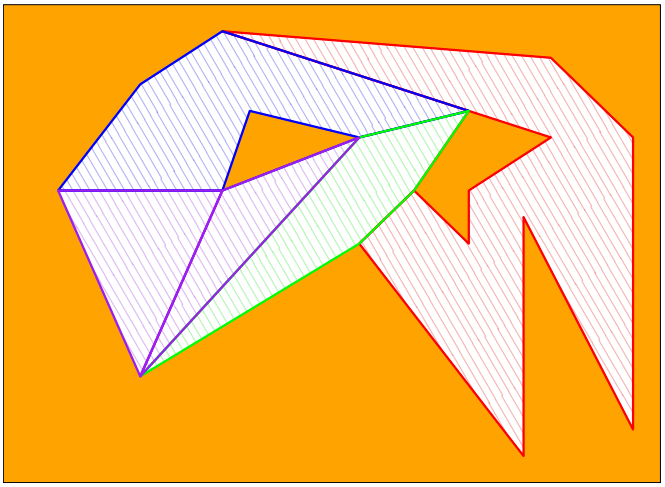
Theorem 5. *Given a set S of sites inside a simple polygon, we can create an $O((n \log^2 n + dn + m)/\epsilon)$ space data structure in $O(dn(\log m + \log n)/\epsilon + (m \log m)/\epsilon + (n \log^2 n)/\epsilon)$ time. With this data structure we can find a $(1 + \epsilon)$ approximate closest site to an arbitrary query point in the polygon in $O(\log^4 n + \log^2 m)$ time, and we can insert and delete sites from the data structure in $O((d(\log m + \log^2 n) + \log^3 n)/\epsilon)$ time.*

5 Polygon with holes

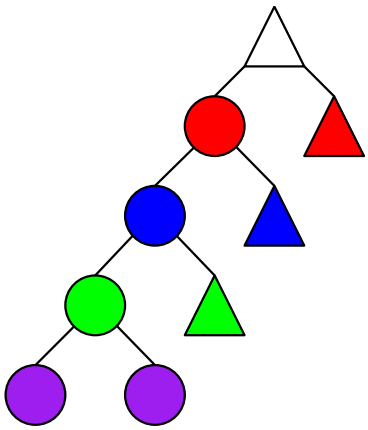
In this section We will discus the second *offline* case, in which we know the locations of the queries we want to do. We can still query the points in any order, and with any temperature range, but we have a finite number of query points, and their locations are known.

5.1 General Approach

We can construct a data structure based on the two-point shortest path algorithm from Thorup [25]. This is an approximation algorithm based on cones, there are k cones, and $\epsilon = \frac{1}{k}$. We will use the separators they construct, and the $O(k)$ number of connections that each vertex has on each separator. Thorup recursively splits the polygon with obstacles in half. The polylines that do this are a union of shortest paths, and are called separators, see Figure 6.



(a) A polygon with holes that is recursively split in two, at the lowest level there are triangles.



(b) A tree that shows how the polygon is split.

Figure 6

Thorup constructs a structure with obstacle vertices as the input, and allows for arbitrary two-point shortest path queries. We will use the vertices, the sites, and the query points as our input. When we do this, the sites and query points are zero-dimensional points, and do not interfere with shortest paths, since you could move an arbitrary small distance around them. By including these points in the construction they influence how the separators are made, they have the property that they have $O(k)$ number of connections on the separators, and the property that going to a neighbouring connection point induces only a stretch of $\frac{1}{k}$, see image 7.

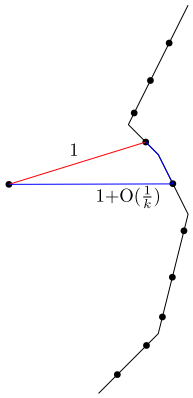


Figure 7: A site with O(k) connections along a separator. If the red path has a length of 1, the blue path has a length of $1 + \frac{1}{k}$. This is true for all neighbouring connection points along the separator.

We create this data structure in $O(((n+m+|Q|)\log^3(n+m+|Q|))/\epsilon^2)$ using $O(((n+m+|Q|)\log(n+m+|Q|))/\epsilon)$ space, and this allows us to do two-point shortest path queries in $O(1/\epsilon^3 + (\log(n+m+|Q|))/(\epsilon \log\log(n+m+|Q|)))$ time, where $|Q|$ is the number of query points.

For ease of notation we will call the union of all these points M, that is $M = n + m + z$. Using this new notation, We create this data structure in $O((M\log^3 M)/\epsilon^2)$ using $O((M\log M)/\epsilon)$ space, and this allows us to do two-point shortest path queries in $O(1/\epsilon^3 + (\log M)/(\epsilon \log\log M))$ time.

This subdivision that is created is a recursive structure with $O(\log M)$ levels. On the lowest level, we have triangles. These triangles are an area of space with no obstacles inside of them.

5.2 Closest Sites on Separators

We want to use the separators and the connections on them to answer our nearest neighbour queries. A query point has $O(k)$ connection points on each separator, $O(M/\epsilon)$ in total. Due to the recursive structure that divides the space in half, we only need to query $O(\log M)$ separators. If every connection point knows which site is the closest, we can compare all the $O((\log M)/\epsilon)$ connection points to find the approximate closest site to the query point.

For a given separator, we are only interested in the $O(k)$ connection points of the sites, and the distances from all the sites to all their connection points. For every point on the separator, we can conclude which site is the closest, by treating every site as a weighted point on the separator, where their distance is their weight.

We can easily see how this works if the separator is a straight line, see image 8. We have a straight line on the x-axis, and all the points have a y-coordinate of their weights. If we

are directly below a point in the x-axis, the distance to the point is its weight. The distance to this point then increases linearly if we move over the x-axis. The distance to a point can be graphed by an absolute function where we translate the function to be on the location of the point. If we do this for all the points, if we draw a line straight up, the first function we intersect is the function of the closest site. We can use this to construct a one-dimensional Voronoi diagram, and now we know for every point on the line what the closest site is.

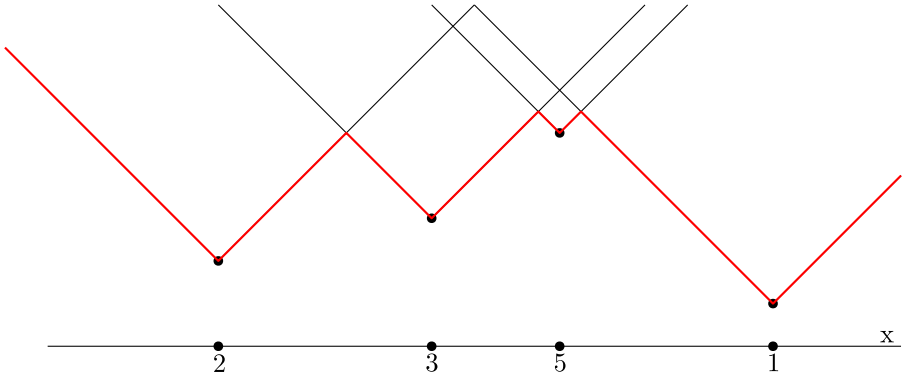


Figure 8: Sites with weights on the x-axis. The sites are placed on the y-location of their weight. With the lower envelope coloured in red.

We can apply the same idea to our separators, however, our separators are not straight lines on the x-axis, to be more precise, our separator are a collection of connected line segments. The same idea still applies, we are interested in the lower envelope of the distance functions of the points. Because the separators are shortest paths themselves, we can straighten them out by walking over them, and place the sites on the x-axis based on their distance from the start of the separator. Using a proof from Hershberger [14] we can find this lower envelope, and we can do this efficiently in a time of $O((n \log(n/\epsilon))/\epsilon)$ time for each separator. In the paper from Hershberger, they find an upper envelope for n line segments. We can translate our problem to this by inverting the y-axis, and by turning the lines into line segments. We know the line segments start at the locations of the sites, and if the end points are put far enough away, the results would be the same as if they were lines. We have a total of $O(M)$ separators, but sites will only influence $O(\log M)$ separators. Sites only influence the separators that divide their space. We need to do a total of $O(n/\epsilon)$ queries to find all the distances to the connections points on a separator. We can write this as the recurrent relation $T(n, M) = T(n_l, \frac{M}{2}) + T(n_r, \frac{M}{2}) + O((n \log(n/\epsilon))/\epsilon + n/\epsilon^4 + (n \log M)/(\epsilon^2 \log \log M))$, where n_l are the sites that go to the *left* side of the separator, and n_r are the sites that go to the *right* side of the separator. If we solve this relation, we get that it takes $O((n \log(n/\epsilon) \log M)/\epsilon + (n \log M)/\epsilon^4 + (n \log^2 M)/(\epsilon^2 \log \log M))$ time to construct all the one dimensional Voronoi diagrams, and they take up $O((n \log M)/\epsilon)$ space. It takes $O(\log(n/\epsilon))$ time to query a single one dimensional Voronoi diagram.

Lemma 6. *Given a polygon with holes P, a set S of sites, a set Q of query points, and the subdivision from Thorup [25], using the vertices, sites and query points as the input. We can create one dimensional Voronoi diagrams on all the separators in $O((n\log(n/\epsilon)\log M)/\epsilon + (n\log M)/\epsilon^4 + (n\log^2 M)/(\epsilon^2\log\log M))$ time, using $O((n\log M)/\epsilon)$ space, where M is the sum of the number of vertices, sites, and query points.*

5.3 Temperature

We can take the temperature value into account during the construction of the one-dimensional Voronoi diagrams. We create binary trees for the temperature, similar to what we do for the naive solution in section 1.3, but instead of creating geodesic Voronoi diagrams on each node, we create a tree for the one-dimensional Voronoi diagrams, which have a size that is only dependant on n and k. We can now still efficiently do queries, by querying a logarithmic number of Voronoi diagrams, but the construction time of these diagrams is not dependant on the number of vertices of the polygon and holes. We can distribute the construction time of the one dimensional Voronoi diagrams into two parts. The construction of the one dimensional Voronoi diagrams themselves, this takes $O((n\log(n/\epsilon)\log M)/\epsilon)$ time, and calculating the distances to the connection points, this takes $O((n\log M)/\epsilon^4 + (n\log^2 M)/(\epsilon^2\log\log M))$ time. We only need to calculate all the distances once during the construction of all the one dimensional Voronoi diagrams in the tree we will build. This means that the first level of the tree has a computation time of $O((n\log(n/\epsilon)\log M)/\epsilon)$, the second level of the tree has a computation time of $O((n\log\frac{n}{2\epsilon}\log M)/\epsilon)$, the third level has a computation time of $O((n\log\frac{n}{4\epsilon}\log M)/\epsilon)$, ..., the last layer has a construction time of $O((n\log M)/\epsilon)$. For every level we can state that the construction time is smaller than $O((n\log(n/\epsilon)\log M)/\epsilon)$. With $O(\log n)$ levels, we get a total construction time of $O((n\log(n/\epsilon)\log n\log M)/\epsilon)$. We can apply the same idea to the space usage, and we get that all the one dimensional Voronoi diagrams use $O((n\log n\log M)/\epsilon)$ space. If we now add back the time to calculate all the distances to the construction time, we get that it takes a total of $O((n\log(n/\epsilon)\log n\log M)/\epsilon + (n\log M)/\epsilon^4 + (n\log^2 M)/(\epsilon^2\log\log M))$ time to construct all the one dimensional Voronoi diagrams. If we add the construction time of the data structure from Thorup, we get a total construction time of $O((n\log(n/\epsilon)\log n\log M)/\epsilon + (n\log M)/\epsilon^4 + (n\log^2 M)/(\epsilon^2\log\log M) + (M\log^3 M)/\epsilon^2)$, and we use a total of $O((M\log M)/\epsilon + (n\log n\log M)/\epsilon)$ space.

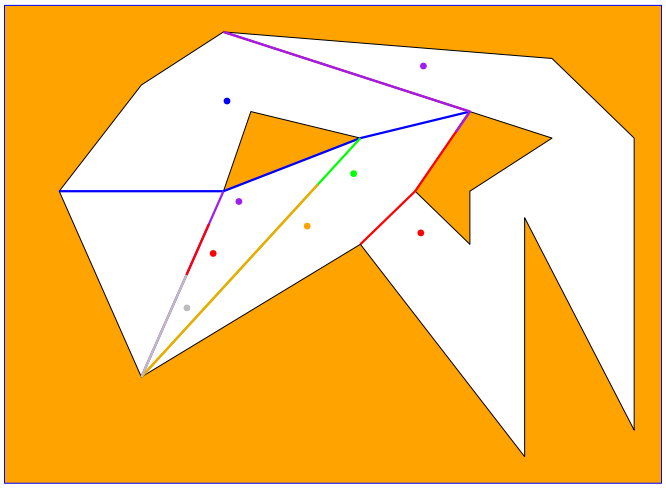


Figure 9: The separators show which sites are the closest on every point. This is shown by giving that piece of separator the same colour as the site.

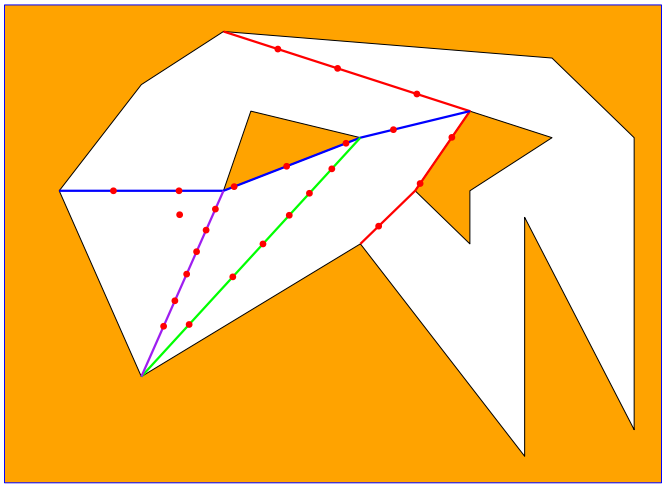


Figure 10: A query point and its $O(k)$ connection points on the separators.

We can answer a query for point a q by querying separators, see Figures 9 and 10. We need to query $O(\log M)$ separators, because we only need to recursively query the separators that are on the same side as q. On every separator, we calculate the distance to the $O(k)$ connection points, this takes $O((\log M)/\epsilon^4 + (\log^2 M)/(\epsilon^2 \log\log M))$ time, and we find the closest site for all the connection points, this takes $O((\log M \log n \log(n/\epsilon))/\epsilon)$ time. This gives us a total query time of $O((\log M)/\epsilon^4 + (\log^2 M)/(\epsilon^2 \log\log M) + (\log M \log n \log(n/\epsilon))/\epsilon)$.

Theorem 7. *Given a set S of sites, and a set Q of query points inside a polygon with holes, we can create an $O((M \log M)/\epsilon + (n \log n \log M)/\epsilon)$ space data structure in $O((n \log(n/\epsilon) \log n \log M)/\epsilon + (n \log M)/\epsilon^4 + (n \log^2 M)/(\epsilon^2 \log\log M) + (M \log^3 M)/\epsilon^2)$ time. With this data structure we can find a $(1+\epsilon)$ approximate closest site to a query point in the polygon in $O((\log M)/\epsilon^4 + (\log^2 M)/(\epsilon^2 \log\log M) + (\log M \log n \log(n/\epsilon))/\epsilon)$ time.*

6 Conclusion

In this study, we looked at an ecological problem in a geometric setting. We represented an environment with a polygon with holes, and species as sets of sites inside this polygon. We process the polygon and the sites such that we can efficiently answer approximate nearest neighbour queries. Additionally, we have looked at more complex cases of the same problem in simpler settings. To the best of our knowledge this is the first approximate algorithm to solve dynamic nearest neighbour searching inside a polygon with holes. The novel use of geometric algorithms contribute to the research in the field of computational geometry, and can serve as an inspiration for future research.

6.1 Future Work

In future research there are two main areas that we could find improvements. We can still make significant progress to lower the time complexity of the case in a simple polygon, and we can work towards solving the *fully dynamic* case for a polygon with holes.

In out solution for the simple polygon, we have a term that is at worst $O(m)$, which is a significant detriment to the performance. We know from Agarwal, Arge, and Staals [3, 1] that we can solve the exact case efficiently, so we should be able to find a more efficient algorithm for the approximate case.

We believe that it is possible to transform our solution for the *offline* case for a polygon with holes, into a solution for the *fully dynamic* case. To achieve this, we need two things. The first thing we need to be able to do, is do efficient insertions and deletions in our one dimensional Voronoi diagrams. Since these diagrams are a lower envelope of simple functions, we believe that we can create efficient dynamic data structures by using the work from Chan [9, 10].

Currently we need to include the sites and the query points in the construction of the data structure from Thorup [25], we currently need to include the sites and query points for two reasons. The first is that the separators themselves are based on all the points, so by removing the sites and query points, the separators would change. The second reason it that during construction, we can efficiently get all the $O(k)$ connection points on all the separators for all the points.

We believe that it is possible to only use the vertices of the polygon and holes to create this data structure, and to efficiently calculate the $O(k)$ connection points of the sites and queries. The separators would now only be based on the vertices, but since the sites and query points are effectively zero dimensional points, we should still get separators that allow us to do nearest neighbour queries. The problem lies in finding these connection points. To know if it is possible, and how to achieve this, we would need a deeper understanding of the algorithms and data structures from Thorup.

If we build the separators using only the vertices, we get that it is possible for a query point q and a site s to be in the same triangle, and their path doesn't cross a separator. Inside each triangle we can use a dynamic nearest neighbour algorithm and data structure for planes, for example, we could use our work from section 3. If we are able to only use the vertices, we could use static Voronoi diagrams in the *offline* case. Our query then splits into two parts, finding the closest site in the same triangle, and finding the closest site on all the separators.

References

[1] Pankaj K. Agarwal, Lars Arge, and Frank Staals. Improved dynamic geodesic nearest neighbor searching in a simple polygon. In Bettina Speckmann and Csaba D. Tóth, editors, *34th International Symposium on Computational Geometry, SoCG 2018, June 11-14, 2018, Budapest, Hungary*, volume 99 of *LIPIcs*, pages 4:1–4:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[2] Arne Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, 1999.

[3] Lars Arge and Frank Staals. Dynamic geodesic nearest neighbor searching in a simple polygon. *CoRR*, abs/1707.02961, 2017.

[4] Boris Aronov. On the geodesic voronoi diagram of point sites in a simple polygon. *Algorithmica*, 4(1):109–140, 1989.

[5] Boris Aronov, Mark de Berg, and Shripad Thite. The complexity of bisectors and voronoi diagrams on realistic terrains. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2008.

[6] Sunil Arya and Theocharis Malamatos. Linear-size approximate voronoi diagrams. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 147–155. ACM/SIAM, 2002.

[7] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[8] Michael T. Burrows, David S. Schoeman, Anthony J. Richardson, Jorge García Molinos, Ary Hoffmann, Lauren B. Buckley, Pippa J. Moore, Christopher J. Brown, John F. Bruno, Carlos M. Duarte, Benjamin S. Halpern, Ove Hoegh-Guldberg, Carrie V. Kappel, Wolfgang Kiessling, Mary I. O'Connor, John M. Pandolfi, Camille Parmesan, William J. Sydeman, Simon Ferrier, Kristen J. Williams, and Elvira S. Poloczanska. Geographical limits to species-range shifts are suggested by climate velocity. *Nature*, 507(7493):492–495, Mar 2014.

[9] Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *J. ACM*, 57(3):16:1–16:15, 2010.

[10] Timothy M. Chan. Dynamic geometric data structures via shallow cuttings. *Discret. Comput. Geom.*, 64(4):1235–1252, 2020.

[11] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[12] Amos Fiat and Gerhard J. Woeginger. Competitive analysis of algorithms. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1996.

[13] Leonidas J. Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.*, 39(2):126–152, 1989.

[14] John Hershberger. Finding the upper envelope of n line segments in o(n log n) time. *Inf. Process. Lett.*, 33(4):169–174, 1989.

[15] John Hershberger. A new data structure for shortest path queries in a simple polygon. *Inf. Process. Lett.*, 38(5):231–235, 1991.

[16] John Hershberger and Subhash Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM J. Comput.*, 28(6):2215–2256, 1999.

[17] Richard Kunze, Franz-Erich Wolter, and Thomas Rausch. Geodesic voronoi diagrams on parametric surfaces. In *Computer Graphics International Conference, CGI 1997, Hasselt and Diepenbeek, Belgium, June 23-27, 1997*, pages 230–237. IEEE Computer Society, 1997.

[18] Didong Li and David B. Dunson. Geodesic distance estimation with spherelets. 2019.

[19] Chih-Hung Liu. A nearly optimal algorithm for the geodesic voronoi diagram of points in a simple polygon. *Algorithmica*, 82(4):915–937, 2020.

[20] Joseph S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Ann. Math. Artif. Intell.*, 3(1):83–105, 1991.

[21] Eunjin Oh and Hee-Kap Ahn. Voronoi diagrams for a moderate-sized point-set in a simple polygon. *Discret. Comput. Geom.*, 63(2):418–454, 2020.

[22] Alejandro Ordonez and John W. Williams. Climatic and biotic velocities for woody taxa distributions over the last 16 000 years in eastern north america. *Ecology Letters*, 16(6):773–781, 2013.

[23] Evanthia Papadopoulou and D. T. Lee. A new approach for the geodesic voronoi diagram of points in a simple polygon and other restricted polygonal domains. *Algorithmica*, 20(4):319–352, 1998.

[24] Pawan Poudel. Computing point-to-point shortest path using an approximate distance oracle. 2008.

[25] Mikkel Thorup. Compact oracles for approximate distances around obstacles in the plane. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, volume 4698 of *Lecture Notes in Computer Science*, pages 383–394. Springer, 2007.