Fortune's algorithm for Voronoi diagrams

Project Report

CSCI 5454

Submitted by:

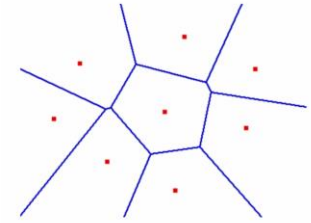
Vibhor Mishra

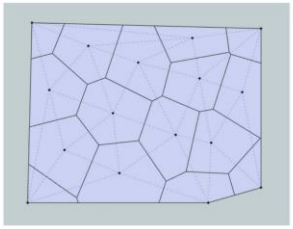
Student Id: 107048343

1. Introduction and History

The Voronoi diagram of a set of sites in the plane, partitions the plane into regions, called Voronoi regions, one to a site. The Voronoi region of a site 's' is the set of points in the plane for which 's' is the closest site among all the sites.

The fortune's sweepline algorithms for the construction of Voronoi diagrams when sites are points is based on the sweepline technique. The sweepline technique conceptually sweeps a vertical/horizontal line in any one particular direction which is normal to the line or across the plane, noting the regions intersected by the line as the line moves. Computing the Voronoi diagram directly with a sweepline technique is difficult, because the Voronoi region of a site may be intersected by the sweepline long before the site itself is intersected by the sweepline. Rather than computing the Voronoi diagram, its geometric transformation is computed. The transformed Voronoi diagram has the property that the lowest point of the transformed Voronoi region of a site appears at the site itself. Thus the sweepline algorithm needs to consider the Voronoi region of a site only when the site has been intersected by the sweepline. It turns out to be easy to reconstruct the real Voronoi diagram from its transformation. In fact in practice the real Voronoi diagram would be constructed, and the transformation computed only as necessary. The sweepline algorithms compute the Voronoi diagram of N sites in time *O(N log N)* and space usage *O(N)*.

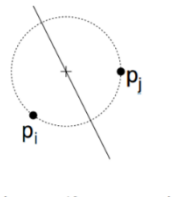




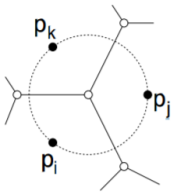
Previous algorithms for Voronoi diagrams fall into two categories. First are incremental algorithms, which construct the Voronoi diagram by adding a site at a time. These algorithms are relatively simple, but have worst-case time complexity of O(n^2). The second category of algorithms are divide-and-conquer algorithms. The set of sites is split into two parts, the Voronoi diagram of each part computed recursively, and then the two Voronoi diagrams merged together. If the sites are points, then they can be split simply by drawing a line that separates the sites into halves. With care, the divide-and-conquer algorithms can be implemented in worst-case time O(N log N). But it was rather complex, and somewhat difficult to understand. The best previously known algorithm for this problem has time complexity O(N log2 N).

2. Terminology

- *Voronoi edges*: Each point on an edge of the Voronoi diagram is equidistant from its two nearest neighbors pi and pj. Thus, there is a circle centered at such a point such that pi and pj lie on this circle, and no other site is interior to the circle. pi pj pi pj pk Figure 59: Properties of the Voronoi diagram.
- *Voronoi vertices*: It follows that the vertex at which three Voronoi cells V(pi), V(pj), and V(pk) intersect, called a Voronoi vertex is equidistant from all sites. Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior.



Voronoi Edge

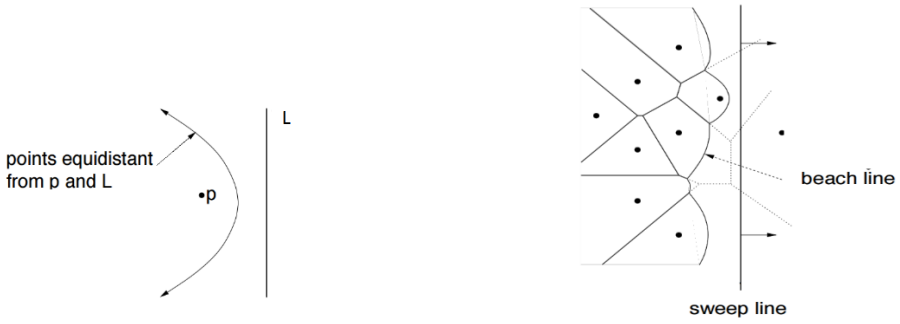


Voronoi Vertex

- *Degree*: If we make the general position assumption that no four sites are cocircular, then the vertices of the Voronoi diagram all have degree three.
- *Convex hull*: A cell of the Voronoi diagram is unbounded if and only if the corresponding site lies on the convex hull. Thus, given a Voronoi diagram, it is easy to extract the convex hull in linear time.
- *Size*: If N denotes the number of sites, then the Voronoi diagram is a planar graph (if we imagine all the unbounded edges as going to a common vertex infinity) with exactly N faces.

2.1. Beach Line:

The set of points q that are equidistant from the sweep line to their nearest site on the left of the sweep line is called the beach line. The portion of the Voronoi diagram that lies on the left of the beach line cannot be changed and doesn't depend on the sites which appear on the right of the sweep line, in the sense that we have all the information that we need to compute. The set of points that are equidistant from a site lying on the left of a vertical line and the line itself forms a parabola that is open on left as shown in the figure. As the site gets closer to the line, the parabola gets thinner. In the degenerate case when the line contains the site the parabola degenerates into a horizontal ray shooting left from the site.

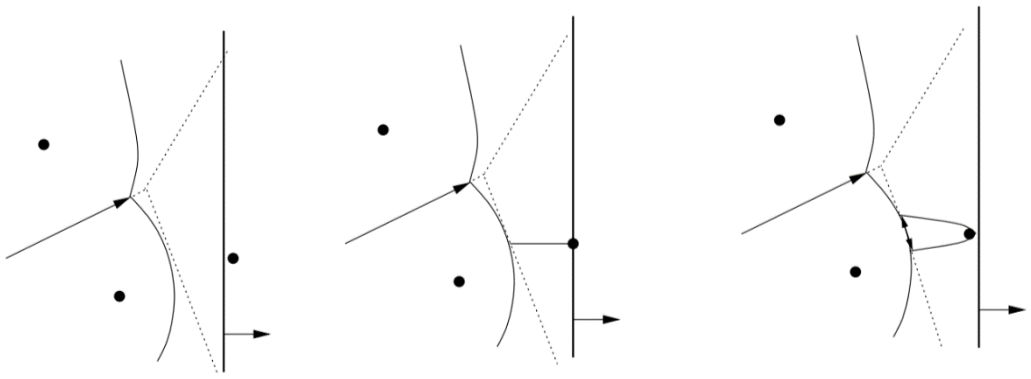


Thus, the beach line consists of the right envelope of these parabolas, one for each site. The parabolas of some sites on the left of the beach line will not touch this envelope and hence will not contribute to the beach line. Because the parabolas are y-monotone, so is the beach line. The vertex where two arcs of the beach line intersect, is called a breakpoint, a point that is equidistant from two sites and the sweep line, and hence must lie on some Voronoi edge. In particular, if the beach line arcs corresponding to sites p_i and p_j share a common breakpoint on the beach line, then this breakpoint lies on the Voronoi edge between p_i and p_j. From this we have the following important characterization.

2.2. Events

There are two types of events that are involved here:

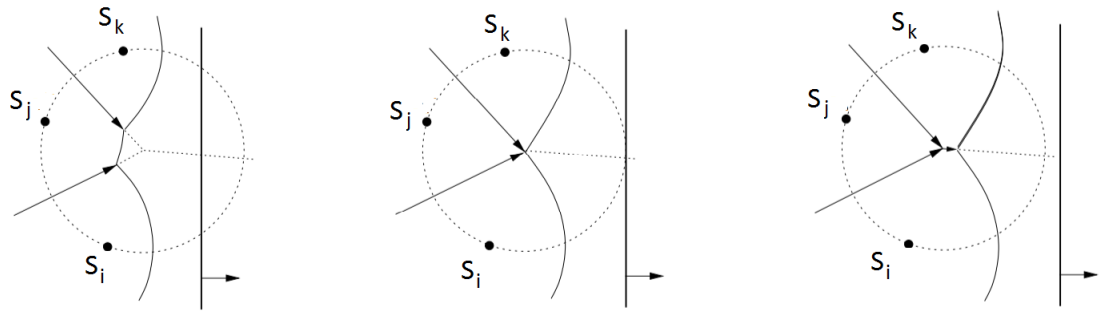
1) Site event: When the sweep line passes over a new site a new arc will be inserted into the beach line. These events are known beforehand i.e. they are not dynamically generated. A site event is generated whenever the vertical sweep line passes over a site. As the sweep line touches the point, its associated parabolic arc will degenerate to a horizontal ray shooting up from the point to the current beach line. As the sweep line proceeds towards the right, this ray will widen into an arc along the beach line. To process a site event, we will determine the arc of the sweep line that lies directly above the new site. We then split this arc of the beach line in two by inserting a new infinitesimally small arc at this point. As the sweep proceeds, this arc will start to widen, and eventually will join up with other edges in the diagram.



Site events

2) Circle/Vertex event: In this event, the parabolic arc disappears and shrinks to zero. At this point, a new vertex, called Voronoi vertex is created. In contrast to site events, these circle events are generated dynamically as the sweep line proceeds in the right direction. Consider three sites as shown in figure below, s_i, s_j and s_k whose arcs appear consecutively on the beach line from bottom to top. Suppose, the circumcircle passing through these three sites lies at least partially on the right of sweep line (i.e. the voronoi vertex is not generated yet) and this circumcircle contains no points(sites) which lie on the right of sweep line which might interrupt the formation of Voronoi vertex.

   At this point, as the sweep touches the rightmost point of the circumcircle i.e. falls as a tangent to the circumcircle on its rightmost point, the center of the circle will be equidistant from all the three sites that lie on its circumference. Thus, all three parabolas from these sites passes through this center point and the contribution of the arc from s_j will disappear from the beach line. This center point of circumcircle will become a Voronoi vertex resulting in just one bisector (s_i, s_k) from two bisectors.



Circle/Vertex event

3. Sweep Line Algorithm's Operations

We have following key operations involved in the algorithm:

Beachline:

1) Given a fixed location of the sweep line and the new site s_i, determine the arc of the beach line that lies on the immediate left of that site. This can be done by a binary search on the breakpoints, which are computed "on the fly".
2) Compute predecessors and successors on the beach line.

3) Insert a new arc *Arc(s_i)* within a given arc *Arc(s_j)* , thus splitting the arc for *Arc(s_j)* into two. This creates three arcs: *Arc(s_j)* , *Arc(s_i)* and *Arc(s_j)* .
4) Delete an arc corresponding to site s_i from the beach line, replacing triples <… *Arc(s_j)* , *Arc(s_i)*, *Arc(s_k)*….> with <… *Arc(s_j)*, *Arc(s_k)* ….>

Event Queue:

1) When three sites on the beachline form a circumcircle and the rightmost point of the circumcircle lies on the right of sweep line, then store that rightmost point of the circle in the queue as a circle event which is linked to those three sites.

Site event: Let s_i be the new site.

1) Advance the sweep line so that it passes through s_i. Apply the above search operation to determine the beach line arc that lies immediately above s_i. Let $s_j$be the corresponding site.
2) Applying the above insert-and-split operation, inserting a new entry for s_i, thus replacing <…, *Arc(s_j)* , *Arc(s_i)*, *Arc(s_k)*….> with <…, *Arc(s_j)*, *Arc(s_k)* ….>.
3) Create a new (dangling) edge in the Voronoi diagram, which lies on the bisector between $s_i$and s_j.
4) Some old triples that involved $s_j$may need to be deleted and some new triples involving $s_i$will be inserted, based on the change of neighbors on the beach line. (The straightforward details are omitted.) Note that the newly created beach-line triple s_j, s_i, s_j does not generate an event because it only involves two distinct sites.

Circle/Voronoi vertex event: Let s_i, s_j, and $s_k$be the three sites that generated this event, from bottom to top.

1) Delete the entry for $s_j$from the beach line status. (Thus eliminating its associated arc.)
2) Create a new vertex in the Voronoi diagram (at the circumcenter of <s_i, s_j, s_k > and join the two Voronoi edges for the bisectors (s_i, s_j) (s_j, s_k) to this vertex.
3) Create a new (dangling) edge for the bisector between $s_i$and s_k.
4) Delete any events that arose from triples involving the arc of s_j, and generate new events corresponding to consecutive triples involving $s_i$and s_k. (There are two of them. The straightforward details are omitted.)

3.1. Data Structures:

1) Priority Queue: All the events are stored in priority queue in the order of increasing x-coordinates of points. This queue stores just the x-coordinates for each site.
2) Binary Search Tree: The beach line is represented in the form of binary search tree to reduce the time complexity in searching the arc which lies directly on the right of a new site discovered by the sweep line. This reduces the complexity to O(nlogn) of the search operation on the tree.

Algorithm

Now, after learning about key operations and data structures involved, we can now formally present the algorithm:

1) Insert all points in a priority queue: *Q* with point having minimum x-cordinate have highest priority and priority of points decreases as x-coordinate of point increases. These points will be processed as site events later.
2) While *Q* is not empty:
    a. *p* <- Pop from *Q*
    b. If *p* is a site (site event):
        i. If root is NULL
            1. Initialize root with *p* and return
        ii. Else
            1. Traverse binary search tree to find the site with arc on the beach line which lies just on the left of *p*.
            2. Divide that arc/region *Arc(q)* on beachline replacing the leaf node with a sub tree representing the new arc and its break points.
            3. Add two half-edge records in the binary search for those arcs.
            4. Check for any circle/Voronoi vertex event and if any exists, insert into *Q*
    c. If *p* is a circle event:
        i. Add vertex to corresponding edge record in BST.
        ii. Delete from T the leaf node of the disappearing arc and its associated circle events in the event queue.
        iii. Create new edge record in the Arc data structure.
        iv. Check the new triplets formed for potential circle events.

4. Implementation

(Please find the code attached as a separate file with this documentation.)

5. Proof of Correctness

Let S be a set of point sites, with unique rightmost site, then to prove the correctness, it would be sufficient to show that the algorithm computes V*(S), where V is the Voronoi diagram and * is the mapping defined by *(z)= (z_x, z_y+ d(z)).

Suppose a region or boundary T or V^* is active if *T* intersects the vertical line through *p* to the top of or at *p* and extends towards left of the line or if *T* intersects the vertical line to the right of *p*, and extends towards the right of the line. Thus, a boundary with minimum point on the line towards the top of *p* is not active and a boundary with maximum point on the line towards the bottom of *p* is not active. We claim statement (iii) following is an invariant of the while loop, and that statements (i) and (ii) following are intermittent invariants of the loop. Specifically, (i) and (ii) are true after the last iteration of the that extracts a particular point *p* from *Q*.

(i) List L contains all regions and boundaries of V^* active at *p*, in the order intersected by the vertical line through *p*.

(ii) If e_{rs} is an edge of *V* (*r* and *s* are arbitrary sites) and e^*_{rs} contains a point lexicographically less than or equal to p then bisector B^*_{rs} is created. If *v* is a vertex of *V* and v^* is lexicographically less than or equal to *p*, then v^* has been marked as a vertex of *V* and as an endpoint of all bisectors containing an edge of V^* incident to v^*.

(iii) If two boundaries are adjacent on L and intersect on the left of the vertical line through *p*, then the intersection is in *Q*.

Intuitively (ii) is true, since the vertical line through the rightmost site *b* intersects *R** at *b* itself and also, since no edge or vertex of *V** contains a point lexicographically less than the minimal site. Similarly, we can say the same about (iii) since no boundary intersects the horizontal line through *b*. We need to prove (i) explicitly and (ii) follows that because we examine the diagram V* in lexicographic order and always correctly create bisectors and label vertices. To prove that (i) is invariant, first note that the set of active regions and boundaries changes only at a site or a vertex of *V**. Now a vertex of *V** must have degree at least three. Hence it sufficient to prove that *L* is updated correctly at a site and at the intersection of boundaries.

First suppose that *p* is a site not lying on a boundary. Furthermore, if *p* lies in the interior of *R** when it was first encountered, then *p* is towards left of edge e_{pq}, and *p* lies on edge e^*_{pq}. Now suppose that *p* is not a site but is the intersection of boundaries. Let boundaries $C_{q_1q_2}$, $C_{q_2q_3}$, … , $C_{q_{m-1}q_m}$, m ≥ 3, all intersect at *p*, in this order from bottom to top, just on the right of p, and suppose that *p* is about to be extracted from *Q* for the first time. By the induction hypothesis (i), *L* contains the sequence $R^*_{q_1}$, $C_{q_1q_2}$, …, $C_{q_{m-1}q_m}$, $R^*_{q_m}$, … ; call this subsequence L_0. As per the arguments above, L_0 should be replaced with $R^*_{q_1}$, $C_{q_1q_m}$, $R^*_{q_m}$, after the last time *p* is extracted from *Q*. It can be noticed that the following assertion is an invariant of the while loop until the last time *p* is extracted from *Q*:

(iv) $R^*_{q_1}$ and $R^*_{q_m}$ are never deleted from L_0, and if R_{q_i}, $C_{q_iq_j}$, R_{q_j}, $C_{q_jq_k}$, R_{q_k} are adjacent on L_0, then $C_{q_iq_j}$ and $C_{q_jq_k}$ intersect at point *p*.

Invariant (iv) is clearly true before *p* is extracted for the first time. Each iteration of the while loop replaces a consecutive pair of boundaries intersecting at *p* with a single boundary. However, since all sites q_1,…, q_m are equidistant from $*^{-1}(p)$, the new boundary intersects its top and bottom neighboring boundaries at *p*, and invariant (iv) is maintained. After *p* is extracted from *Q* for the last time, only R_{q_1}, $C_{q_1 q_m}$, R_{q_m} remain, and invariant (i) is established.

We see from the proof of that it does not matter in what order multiple events at a single site are processed. That is why Algorithm can handle degeneracies in the placements of the sites without being explicitly coded to do so.

6. Analysis: Space and time complexity

6.1. Runtime Complexity

The following functions are used in the code: [Notations: In O(c), c is any constant]

1) *main()* : In main function, we add all points to the priority queue: *events* which is declared as global variable. At this initial point, we know all the site events which corresponds to the points in Voronoi diagram. What we don't know is the number of circle/Voronoi vertex events that might occur later as the sweep line moves from left to right. Let's have a look at the following property to get an idea about future circle events:
    a) If there are more than 3 points in Voronoi diagram, it will contain maximum ***2n-5*** vertices.
    For sites lying in the line, it is very much evident. Suppose, they don't lie in a line. From Euler's formula,
        ⇨ V-E+N = 2

    where V, E and N represent number of Voronoi vertices, number of edges and number of sites respectively. As Voronoi diagram contain infinite edges, let's create a new vertex infinity and collect all edges to it.

        ⇨ (V+1) - E + N = 2 …. (i)

    We know, that every voronoi vertex has degree of atleast 3 vertices, we can write it as:

        ⇨ 3∗(V+1) ≤ (2∗E)
        ⇨ V ≤ (2∗N – 5) …..using (i)

    Therefore, using this result, we can compute the upper bound on priority queue operations.[2]:

    a) Insert: Θ(1)
    b) Find-max and pop: Θ(log N') = Θ(log (2∗N – 5)) = Θ(log N)

    Thus the complexity of iterating through priority queue in this whole process is: ***Θ(N log N)*** as the pop operation will be performed N times.

2) siteEvent(pt): This function processes the site events poped from the priority queue in the order of increasing x-coordinates. This function implements binary search tree (BST) for the beach line and manipulates the data structures accordingly. It consists of following operations:
    a. Search for the immediate left arc to the site by traversing the tree from root node. This search operation on BST of arcs has the average-case complexity of ***Θ (log N)*** time and worst case complexity of ***O(N^2)***.
    b. Once it finds the arc, it divides it into two, introducing extra level in the tree with two non-leaf nodes, thus modifying and setting the left and right child nodes and, setting left and right neighbors respectively. This operation takes constant time of ***O(1)***.
    c. Once this arc is divided into two, it then checks for circle event possibility for the triples as explained in previous section through the call to *checkForCircleEvent* () which also takes up constant time: ***O(1)***.

3) circleEvent(event): This function handles the Voronoi vertex event or clircle event where the arc corresponding to middle site out of three sites vanishes at the center of circumcircle formed of those three sites, resulting in Voronoi vertex.
    a. The '*event'* passed to this function already has corresponding arc information and using which the corresponding arc is deleted from BST that represents beachline.
    b. After deleting the arc, again we need to check for circle event for the resulting two sites with their two neighbors on left and right. The complexity of this operation is constant as *checkForCircleEvent()* is called which takes up constant time: ***O(1)***

4) *checkForCircleEvent(arc, x0):* This function checks whether the *'arc'*, it's left neighbor and right neighbor forms a circle event or not. In this, a call to *getCircle* is made to do the calculation. Thus, adding the event if *getCircle* returns true else return. Therefore, the complexity of this function is in constant time: ***O(1)***

5) getCircle(a, b, c, o): This function takes three points as input for which we check if they form a circle or not. The calculation for the same is done using O'Rourke Algorithm which computes the circle in constant time in fixed number of steps. Thus, its complexity is: ***O(1)***.

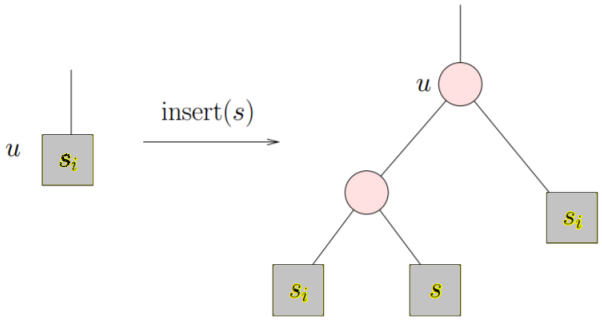
6) doPtIntersectParabola (p, arc, res): Here '*p*' is the point and '*arc*' is region on beackline for which we need to check whether the arc lies on the immediate left of the beachline or not.
    a. To perform this calculation, we fetch left and right neighbor which takes *O(1)* as we store the address of neighbors of sites forming arc on beachline in the '*Arc*' datastructure.
    b. We then call *getParaIntersectionPoint()* with the points of current arc and its left arc and then again with the current arc and its right arc. The purpose of this is to find the location of intersection of the two arcs and compare the resultant y-coordinate with the y-coordinate of point '*p'* to check if the point lies between the two intersections points or not. With this we can make sure if the points lies directly on the right of the middle arc ('*arc*' passed to this function) or not. This calls too just take constant time of ***O(1)***.

7) *getParaIntersectionPoint(p0, p1, px):* As explained in above point, this function computes the intersection of two parabolas in constant time by solving two parabolas equation using quadratic equation which is computed in constant time.

8) *completeEdges():* This function is called at the last when the sweep line finishes sweeping through all the sites/points in the Voronoi diagram. It computes the endpoints of dangling edges which belongs to the sites on the beach line. When the sweep line reaches the extreme right toward the end beyond which there are no sites, the sites forming arcs on beachline will have their bisector extends to infinity beyond the bounding region. So, starting from leftmost arc on beachline, one at a time the intersection of that arc with its right neighbor is calculated that will lie surely outside the bounding box to complete the dangling edge of that leftmost site. We require the following operations to complete the dangling edges:
   a. Traverse every node of BST to reach its leaves.
   b. Compute the intersection point and complete the dangling edges.

   To compute its complexity, we must know about certain properties of the beach line and Voronoi vertices:

   1) Maximum number of arcs on beachline can be at most '***2∗N-1***' since each new point can result in creating one new arc, and splitting an existing arc causing net increase of 2 arcs per point except the first point. And as these site events are known in advance we can know the maximum leaves that can exist in the BST.
   2) If there are ***N*** leaf nodes in total forming arcs on beachline, then there will be maximum of ***N-1*** non-leaf nodes in the tree. This is quite intuitive because a non-leaf node is formed only when an existing leaf node is splitted into three different arcs on beachline. Thus, splitting of each leaf node results in the formation of one non-leaf node



Inserting s into leaf u containing s_i

Thus, from these observations, we can compute the time complexity of this step:

Maximum number of nodes in tree = (leaf nodes) + (non-leaf nodes)

= (2∗N -1) + (2∗N -1 -1) = 4∗N – 3

Therefore, the time complexity to traverse all the nodes in BST for this step would be :

⇨ O(4∗N-3) = ***O(N)***

6.2. Space Complexity:

1) Priority queue: As we saw above under '*main()*' function that the Voronoi diagram can have at most '*2∗N-5*' vertices, we can compute its space complexity as:

   Space Complexity = space required to store sites + space required to store vertices

   = O(N) + O(2∗N-5) = ***O(N)***

2) Binary Search Tree: The beachline in the algorithm is represented in the form of binary search tree of '*Arc*' class objects in the implementation code. As we know the maximum number of nodes that can exist in BST here, its space complexity can be computed as:

   Space complexity = O(4∗c∗N – 3) = O(c∗N) = ***O(N)***

   where c is the sum of the sizes of Arc object and it components which includes Event, Point and Segment objects as their size can be represented by a constant value.

6.3. Overall Complexity

Thus, on summing up the complexities of each individual functions, we get the following overall time and space complexity:

Average Time complexity = Θ (N) ∗ (Θ (log N) + Θ (log N)) = ***Θ (N log N)***

Worst time complexity = O(N) ∗ (O(log N) + $O(N^2)$) = ***O(N^2)***

where Θ (N) is for iterating sites from priority queue and Θ (log N) to pop element with highest priority from queue and another Θ (log N) for searching Arc in BST. Rest all operations have technically constant time complexity.

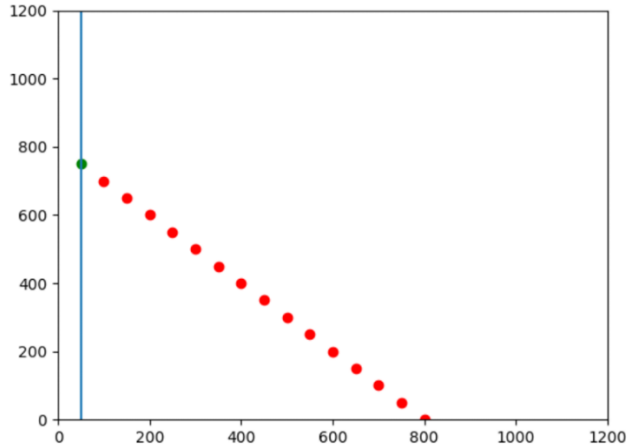
Space Complexity = O(N) + O(N) = ***O(N)***

7. Performance bounds

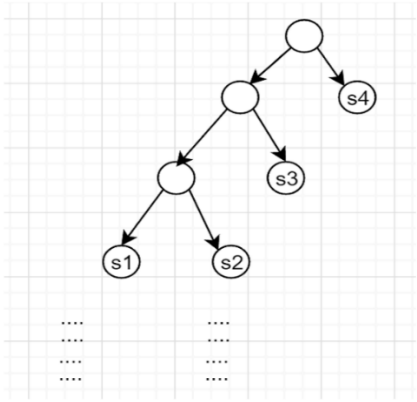
The main operation that contributes much to the complexity is search of the arc in binary search tree. Most of the performance is totally based on how efficiently the arc can be searched in the tree. The main key-point here is the height of the tree. More the height of the tree, more will be the search time.

7.1. Worst-case performance

The algorithm will perform worst when the sites/points are arranged as shown in the following figure:



Red Dots are undiscovered sites and line in Blue is the Sweep Line



BST for the arrangement of sites shown on left

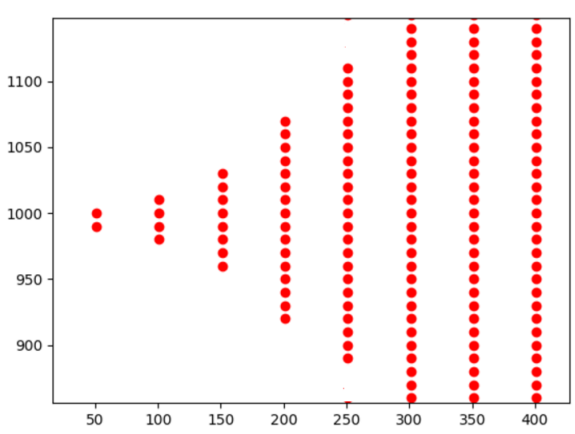
In this kind of arrangement, the sites from left to right are arranged in downward fashion. Such an arrangement will result in binary tree growth only in left direction, thus increasing the depth to N-1, number of sites. As the sweep line moves towards the right, any new site it encounters will be added to the leftmost arc on beachline for which it has to traverse through all N-1 nodes. In such a case, the time complexity of search in BST increases to ***O(N)*** *from O(log N)*.

Time taken to generate Voronoi diagram for 1000 such points = *6.56 secs*

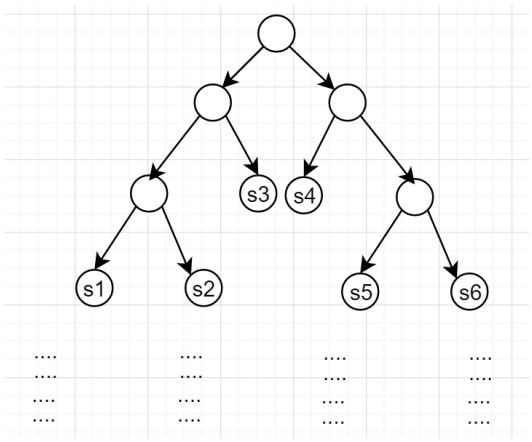
Time taken by randomly generated input of 1000 points = *0.86 secs*.

7.2. Best-case performance:

The algorithm will perform best when the sites are arranged in a pattern that the height of the tree remains balanced and optimum such that maximum nodes traversed for search remains O(log N).



Arrangement of Sites/Points



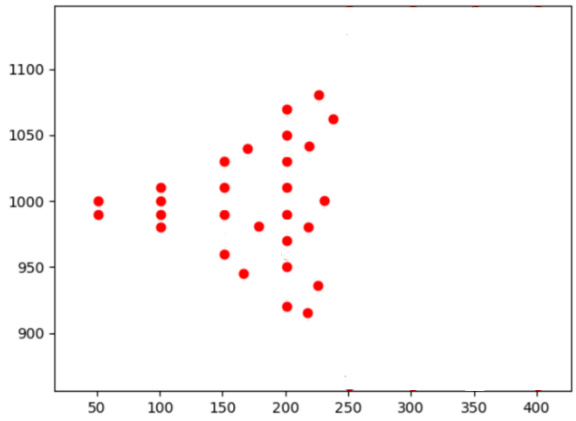
BST with log N height

Time taken to generate Voronoi diagram for 1000 such points = *0.256 secs*

Time taken by randomly generated input of 1000 points = *0.464 secs*.

7.3. Average-case performance:

As per the average case performance of BST[3], $P(N) \leq 1 + 1.4(\log N)$, where P(N) is the average path length in a BST with N nodes i.e. average number of nodes on the path from the root to a node. Intuitively, the uniformly generated random input of points/sites will most fit in this average case. As per the expression, the height of the tree should remain within the limit of *1.4(log N)* bound.



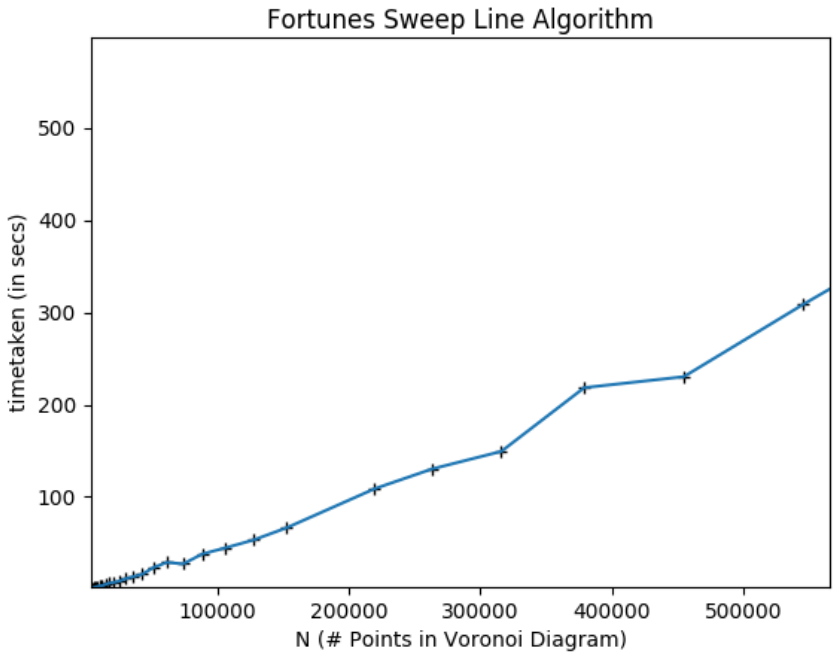
In this example of 28 points, there is no instance of points going down continuously to deep level like in the worst case and also, the height at each level doesn't exceed the given bound too.

Time taken to generate Voronoi diagram for 1000 such points = *0.987 secs*

Time taken by randomly generated input of 1000 points = *0.879 secs*.

8. Performance on random inputs

The following plot shows the performance of the code on random inputs of sites/points with number of points/sites ranging from (10 to 500,000). As the plot depicts, the runtime is better than O(N^2) and is very much close to O(N log N).



Note: This graph is generated by calling '*PerformanceAnalyzer()*' function in the code rather than calling '*main()*' in the end of the python script file.

9. Applications

Voronoi diagram has ample number of applications in various fields[4] . Following mentioned are the few of those:

1) ***Image compression***: For lossy compression when K-means algorithm is applied to an image, the resulting clusters are nothing but Voronoi diagram whose centroids (sites/points in Voronoi diagram) can be stored instead of all pixel's intensity values, thus resulting in maximum compression. Also, as the process of expansion of image from centroids or sites is linear in time, this proves to be an efficient process of uncompression.
2) In **machine learning**, Voronoi diagrams are used to do 1-NN classifications.

3) **Largest Empty Circle** also known as Toxic Waste Dump Problem.
4) In **mining**, Voronoi polygons are used to estimate the reserves of valuable materials, minerals, or other resources. Exploratory drillholes are used as the set of points in the Voronoi polygons.
5) In **biology**, Voronoi diagrams are used to model a number of different biological structures, including cells and bone microarchitecture. Indeed, Voronoi tessellations work as a geometrical tool to understand the physical constraints that drive the organization of biological tissues.
6) In **hydrology**, Voronoi diagrams are used to calculate the rainfall of an area, based on a series of point measurements. In this usage, they are generally referred to as Thiessen polygons.
7) In **ecology**, Voronoi diagrams are used to study the growth patterns of forests and forest canopies, and may also be helpful in developing predictive models for forest fires.

10. References

1) Steven Fortune. "A Sweepline Algorithm for Voronoi Diagrams". Algorithmica (1987) 2:153-174
2) Priority Queue time complexities: https://en.wikipedia.org/wiki/Priority_queue
3) Jeffrey Scott Vitter and Philippe Flajolet "Average-Case Analysis of Algorithms and Data Structures", http://www.ittc.ku.edu/~jsv/Papers/ViF90.AAA.pdf
4) Applications of Voronoi diagrams: https://en.wikipedia.org/wiki/Voronoi_diagram#Applications

11. Disclaimer

This report is a consolidation of various research works listed in the References section. My individual contribution is mainly just in implementing the algorithm and the experimental analysis of the same. So, this document might contain inferences from any of those research papers or, in some parts, excerpts of those. The credit goes to the original work in such cases.