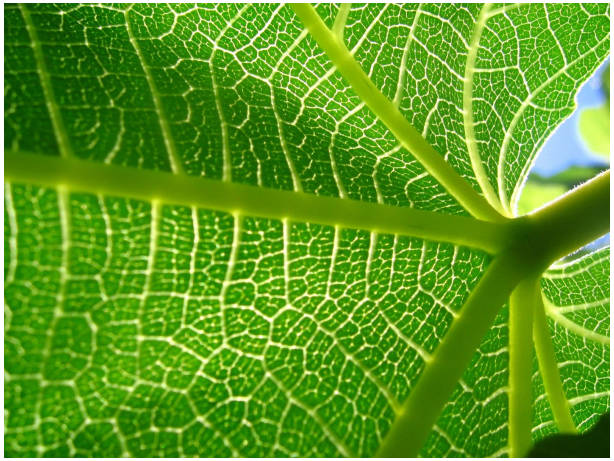
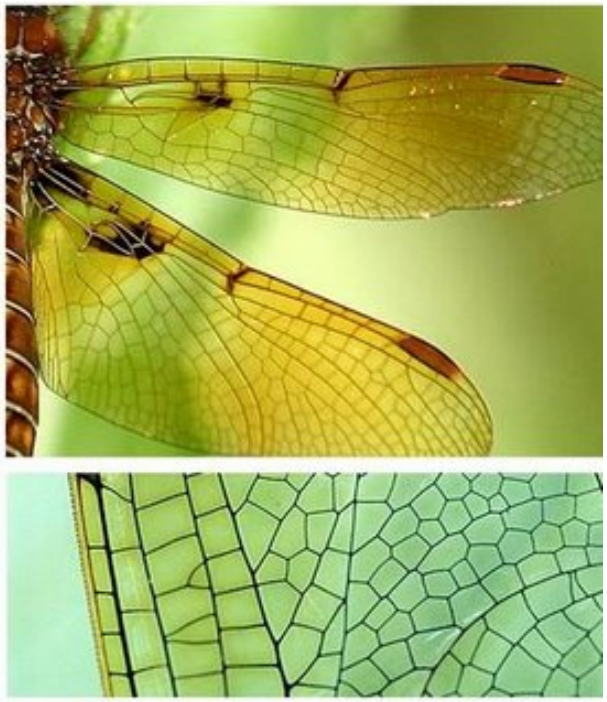
Voronoi diagrams

J.-F. Remacle
Université catholique de Louvain

October 15, 2013

Voronoi diagrams





The Supermarket Problem

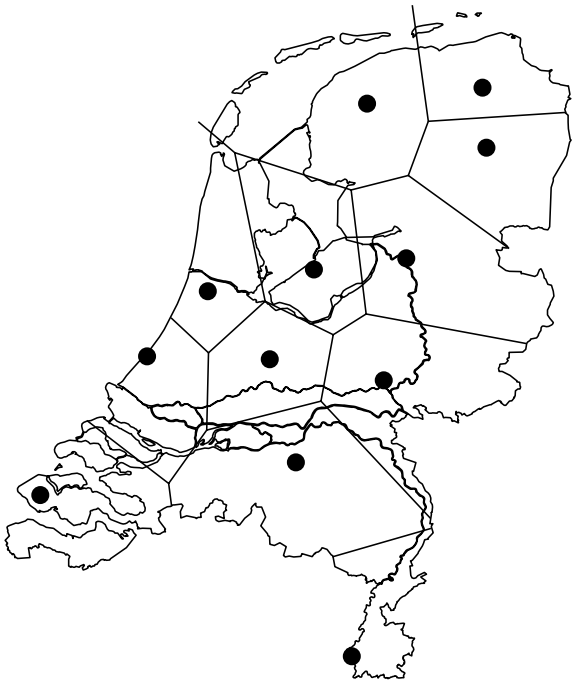
Suppose we aim at distributing N supermarkets in a given region in such a way that it optimizes the number of customers they will attract.

For this you have to model the behavior of your potential customers: how do people decide where to do their shopping? To study this question we make the following simplifying assumptions:

- the price of a particular good or service is the same at every site;
- the cost of acquiring the good or service is equal to the price plus the cost of transportation to the site;
- the cost of transportation to a site equals the Euclidean distance to the site times a fixed price per unit distance;
- consumers try to minimize the cost of acquiring the good or service.

The Supermarket Problem

The assumptions in the model induce a subdivision of the total area under consideration into regions such that the people who live in the same region all go to the same site. Our assumptions imply that people simply get their goods at the nearest site. The model where every point is assigned to the nearest site is called the Voronoi assignment model. The subdivision induced by this model is called the Voronoi diagram of the set of sites.



Definitions

Consider a set P of N distinct points $P := \{\mathbf{p}_1, \ldots, \mathbf{p}_N\}$ in the plane R^2. These points are called the sites. We define the Voronoi diagram of P as the subdivision of the plane into N cells, one for each site in P, with the property that a point $\mathbf{q}$ lies in the cell corresponding to a site $\mathbf{p}_i$ if and only if

$$\text{dist}(\mathbf{q}, \mathbf{p}_i) < \text{dist}(\mathbf{q}, \mathbf{p}_j) \quad \forall \mathbf{p}_j \in P, \quad j \neq i.$$

Here, $\text{dist}(\mathbf{a}, \mathbf{b})$ is the euclidian distance between $\mathbf{a}$ and $\mathbf{b}$.

We denote the Voronoi diagram of P by $\text{Vor}(P)$. The cell of $\text{Vor}(P)$ that corresponds to a site $\mathbf{p}_i$ is denoted $\mathcal{V}(\mathbf{p}_i)$; we call it the Voronoi cell of $\mathbf{p}_i$.

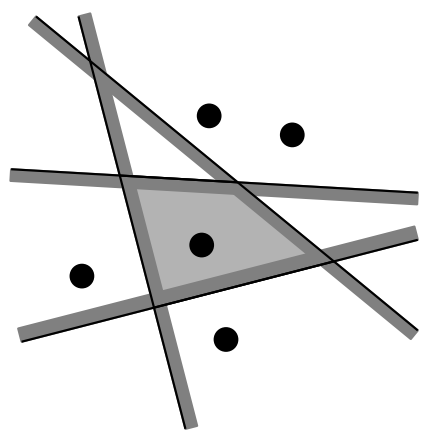
Orthogonal Bissectors

Consider two points $\mathbf{p}$ and $\mathbf{q}$. We define the orthogonal bisector of $\mathbf{p}$ and $\mathbf{q}$ as the perpendicular bisector of the line segment $\mathbf{pq}$. This bisector splits the plane into two half-planes.

We denote the open half-plane that contains $\mathbf{p}$ by $h(\mathbf{p}, \mathbf{q})$ and the open half-plane that contains q by $h(\mathbf{q}, \mathbf{p})$. Notice that $\mathbf{r} \in h(\mathbf{p}, \mathbf{q})$ if and only if $\text{dist}(\mathbf{r}, \mathbf{p}) < \text{dist}(\mathbf{r}, \mathbf{q})$.

We have

$$\mathcal{V}(\mathbf{p}_i) = \bigcap_{1 \leq j \leq N, j \neq i} h(\mathbf{p}_i, \mathbf{p}_j).$$



Complexity of Voronoi Diagrams

Since there are N sites and each Voronoi cell has at most $N-1$ vertices and edges, the complexity of Vor(P) is therefore at most quadratic.

If all the sites are collinear then Vor(P) consists of $N-1$ parallel lines. Otherwise, Vor(P) is connected and its edges are either segments or half-lines.

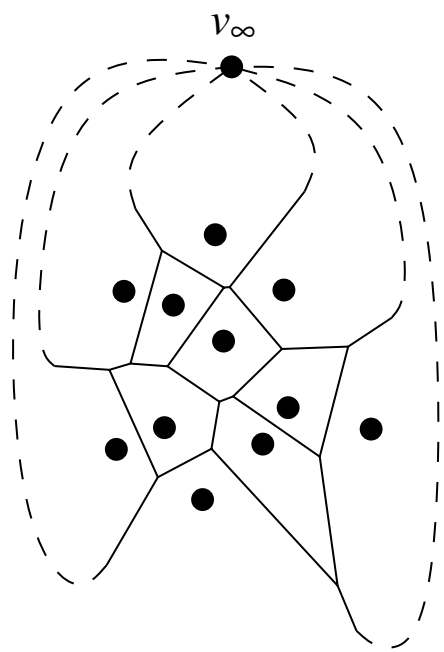
Theorem : For $N \geq 3$, the number of vertices n_v in the Voronoi diagram of a set of N sites in the plane is at most $2N-5$ and the number of edges n_e is at most $3N-6$.

Euler's formula : for any connected planar embedded graph with n_v nodes, n_e edges and n_f faces, the following relation holds:

$$n_v - n_e + n_f = 2.$$

Complexity of Voronoi Diagrams

Create an augmented graph that is connected:



Euler's formula applied to that graph gives:

$$(n_v + 1) - n_e + N = 2$$

Complexity of Voronoi Diagrams

Every edge in the augmented graph has exactly two vertices, so if we sum the degree of all vertices, we get twice the number of edges. Moreover, the degree of one vertex is at least 3 because $\mathcal{V}(P)$ is composed of edges or half-lines. Then,

$$2n_e \geq 3(n_v + 1).$$

This leads to

$$3(n_v + 1) = 6 + 3n_e - 3N \leq 2n_e \quad \rightarrow \quad n_e \leq 3N - 6.$$

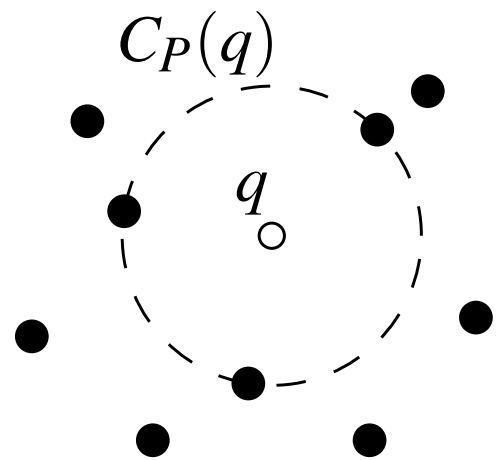
and

$$2(3N - 6) \geq 3(n_v + 1) \quad \rightarrow \quad n_v \leq 2N - 5.$$

We conclude that the complexity will never be quadratic.

Empty circles

For a point $\mathbf{q}$ we define the largest empty circle of $\mathbf{q}$ with respect to P, denoted by $C_P(\mathbf{q})$, as the largest circle with $\mathbf{q}$ as its center that does not contain any site of P in its interior.

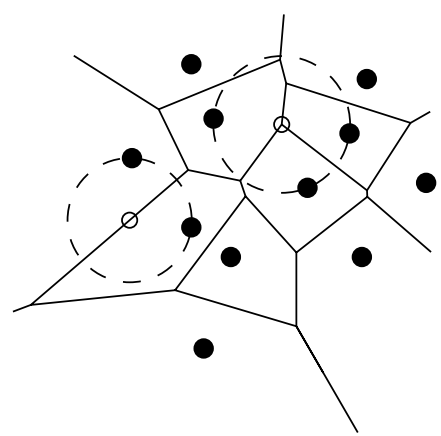


Empty circles

Theorem : For the Voronoi diagram Vor(P) of a set of points P the following holds:

- A point $\mathbf{q}$ is a vertex of Vor(P) if and only if its largest empty circle $C_P(\mathbf{q})$ contains three or more sites on its boundary.

- The bisector between sites $\mathbf{p}_i$ and $\mathbf{p}_j$ defines an edge of Vor(P) if and only if there is a point $\mathbf{q}$ on the bisector such that $C_P(\mathbf{q})$ contains both $\mathbf{p}_i$ and $\mathbf{p}_j$ on its boundary but no other site.



Computing Voronoi Diagrams

Many methods available :

- Explicit computation through intersection of half planes (`voro++`);
- Iterative computation (related to Sibson or natural interpolation);
- Sweep line (Fortune's algorithm);
- Dual of the Delaunay triangulation;
- ...

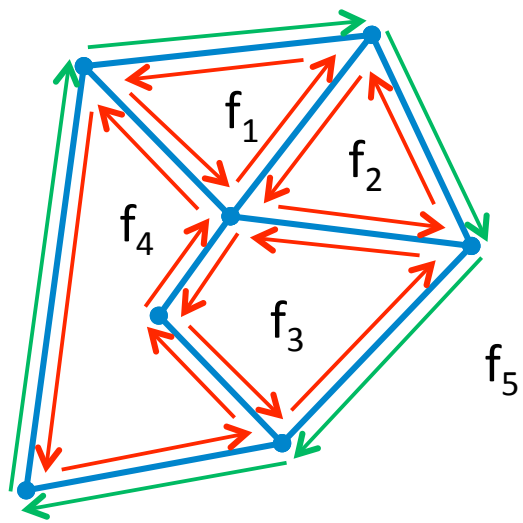
Doubly connected edge list

(Partial) Voronoi diagram: The partial Voronoi diagram that has been constructed so far will be stored in a DCEL. There is one technical difficulty caused by the fact that the diagram contains unbounded edges. To handle this we will assume that the entire diagram is to be stored within a large bounding box. (This box should be chosen large enough that all of the Voronoi vertices fit within the box.)

The doubly connected edge list (DCEL) is a data structure to represent an embedding of a planar graph in the plane and polytopes in 3D. This data structure provides efficient manipulation of the topological information associated with the objects in question (vertices, edges, faces). It is used in many algorithms of computational geometry to handle polygonal subdivisions of the plane, commonly called planar straigh-line graphs (PSLG).

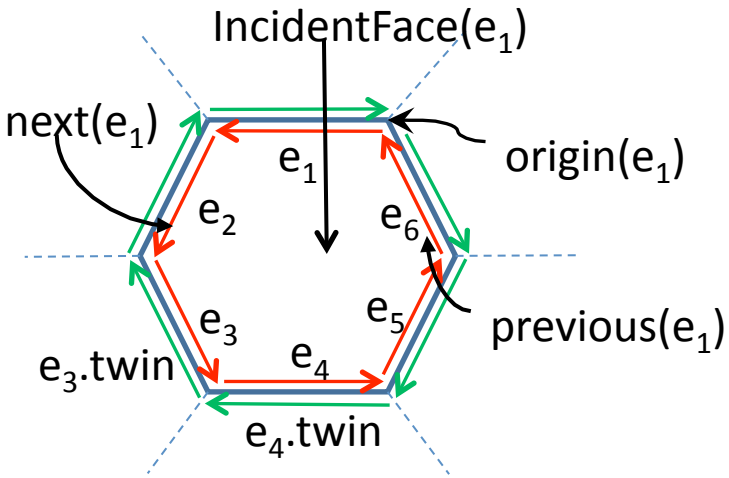
Doubly connected edge list

Link together three sets of records: vertices, edges and faces.



Edges are oriented counterclockwise inside each face. Since an edge borders two faces, each edge is replaced by two half-edges, one for each face.

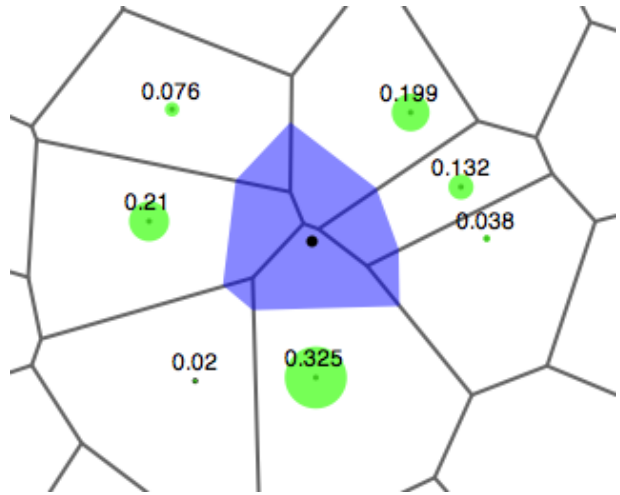
Doubly connected edge list



The vertex stores its coordinates as well as a pointer to one of the half-edges it is connected to. The face stores a pointer to some of its half-edges. The half-edges store pointers to its origin vertex, to its twin, to the face to it is left, to the next and the previous half-edges on its incident face.

Operations: walk around a face in CCW order, access a face adjacent to a given one, visit all edges around a vertex.

Sibson interpolation

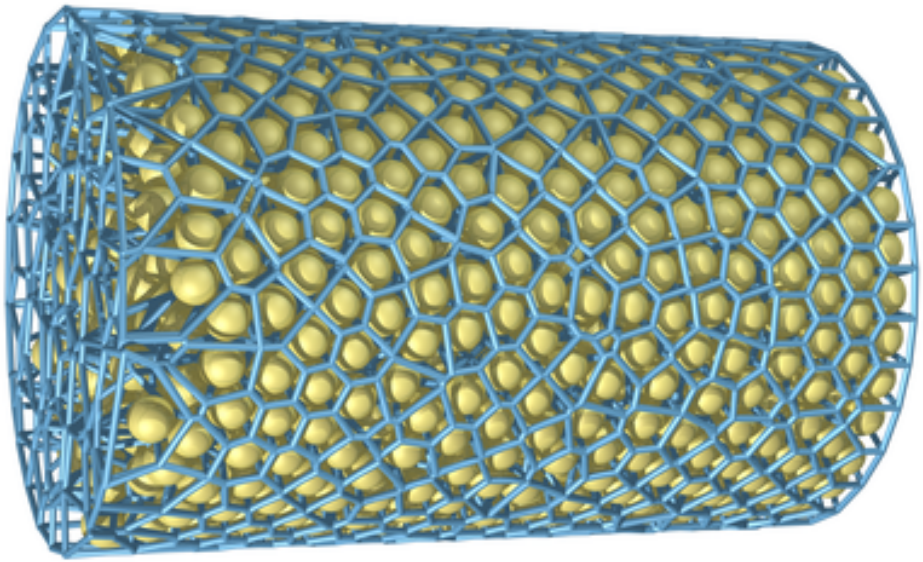


Provides smooth approximation F to $f(x, y)$:

$$F(x, y) = \sum_{i=1}^{n} w_i f(x_i, y_i)$$

where the w_i's are the weights and $f(x_i, y_i)$ are the known data at (x_i, y_i). The weights, w_i, are calculated by finding how much of each of the surrounding areas is "stolen" when inserting (x, y) into the tessellation.

Explicit Computation of Voronoi Diagrams



- For each site $\mathbf{p}_i$, compute the common intersection of the half-planes $h(\mathbf{p}_i, \mathbf{p}_i)$, with $j \neq i$, using an algorithm that is $O(NlogN)$ time per Voronoi cell (as we will see)

- This leads to an $O(N^2 logN)$ algorithm to compute the whole Voronoi diagram.

Intersection of half-planes

A convex polytope is defined as the intersection of N half-planes in R^d. Let us devise a divide and conquer method for effectively computing the intersection of N half-planes in 2D.

- If $N = 1$, then just return this half-plane as the answer.
- Split the N half-planes of P into subsets P_1 and P_2 of sizes $N/2$.
- Compute the intersection of P_1 and P_2, each by calling this procedure recursively. Let C_1 and C_2 be the results.
- Intersect the convex polygons C_1 and C_2 (which might be unbounded) into a single convex polygon C, and return C.

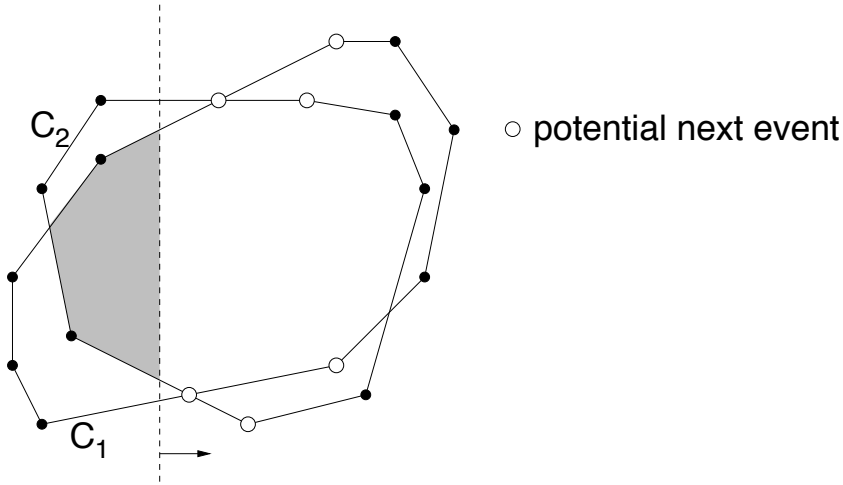
Intersection of two convex polygons

Assume that it is possible to compute the intersection of N line segments in $O((N+I)\log N)$ time, where I is the number of intersecting pairs.

Two convex polygons cannot intersect in more than $I = O(N)$ pairs. This follows from the observation that each edge of one polygon can intersect at most two edges of the other polygon by convexity.

This would given $O(N \log N)$ algorithm for computing the intersection and an $O(N \log^2 N)$ solution for the problem of intersection of half-plane, which is not as good as we would like.

Intersection of two convex polygons

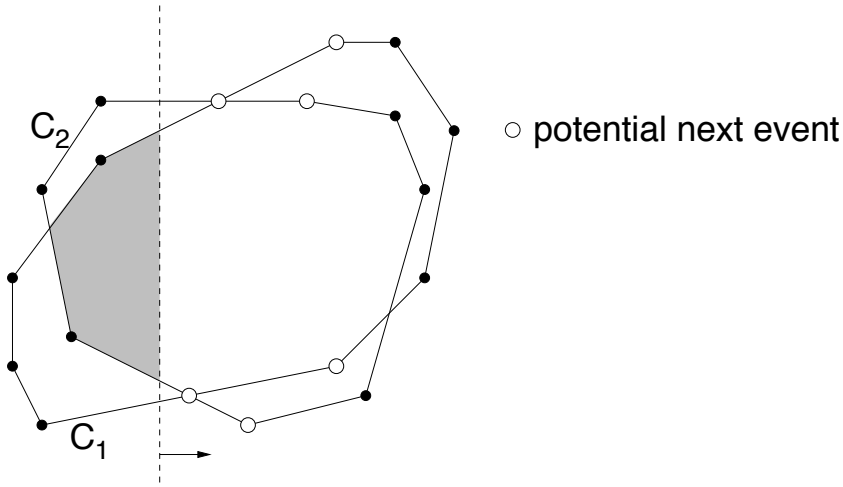


We perform a left-to-right plane sweep to compute the intersection.

By convexity, the sweep line intersects each convex polygon C_i in at most two points, and hence, there are at most four points in the sweep line status at any time.

Thus we do not need a dictionary for storing the sweep line status, a simple 4-element list suffices.

Intersection of two convex polygons



Also, our event queue need only be of constant size. At any point there are at most 8 possible candidates for the next event, namely, the right endpoints of the four edges stabbed by the sweep line and the (up to four) intersection points of these upper and lower edges of C_1 with the upper and lower edges of C_2.

Since there are only a constant number of possible events, and each can be handled in $O(1)$ time, the total running time is $O(N)$.

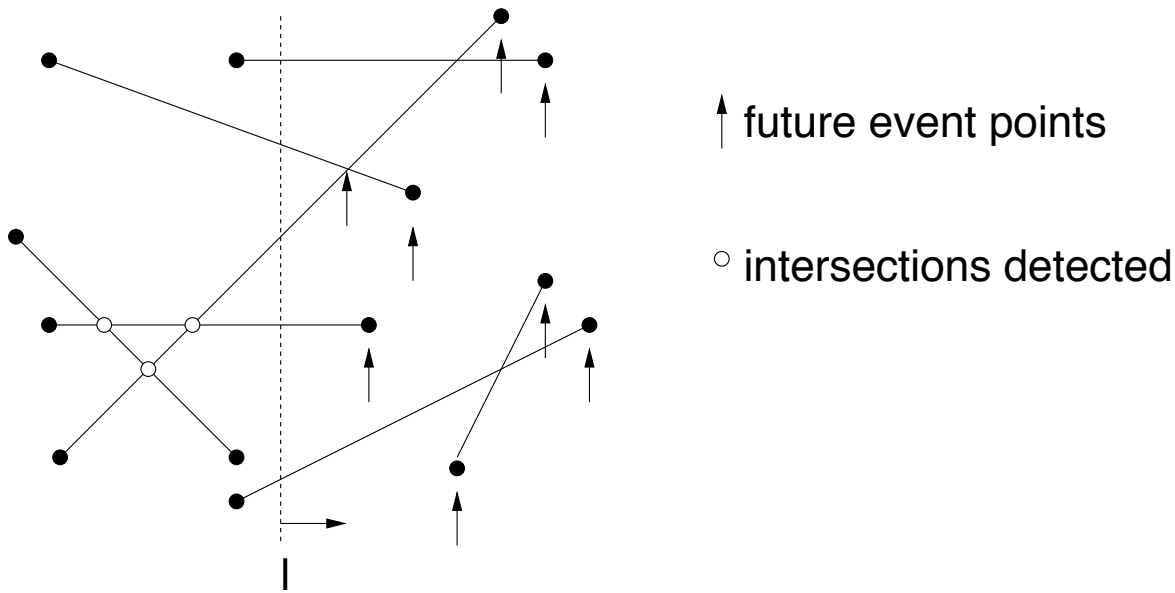
Intersection of line segments

The problem: Given n line segments in the plane, report all points where a pair of line segments intersect.

Observe that n line segments can intersect in as few as 0 and as many as $\binom{n}{2} = O(n^2)$ different intersection points.

We could settle for an $O(n^2)$ algorithm, claiming that it is worst-case asymptotically optimal, but it would not be very useful in practice.

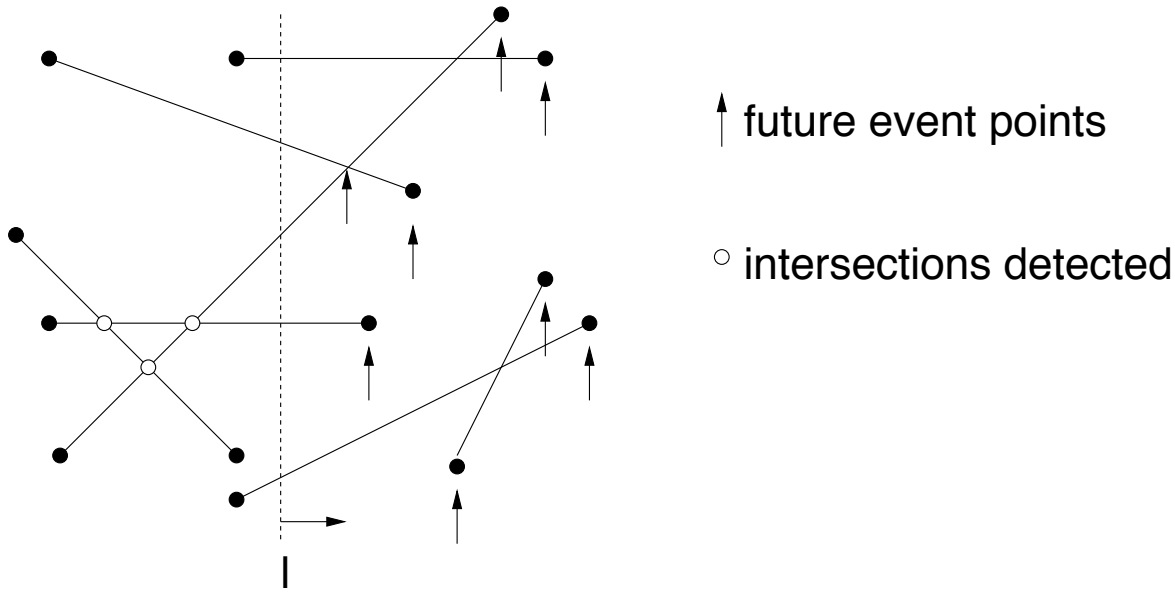
Intersection of line segments



Sweep line: We will simulate the sweeping of a vertical line l, called the sweep line from left to right. We will maintain the line segments that intersect the sweep line in sorted order (say from top to bottom).

Events: Although we might think of the sweep line as moving continuously, we only need to update data structures at points of some significant change in the sweep-line contents, called event points.

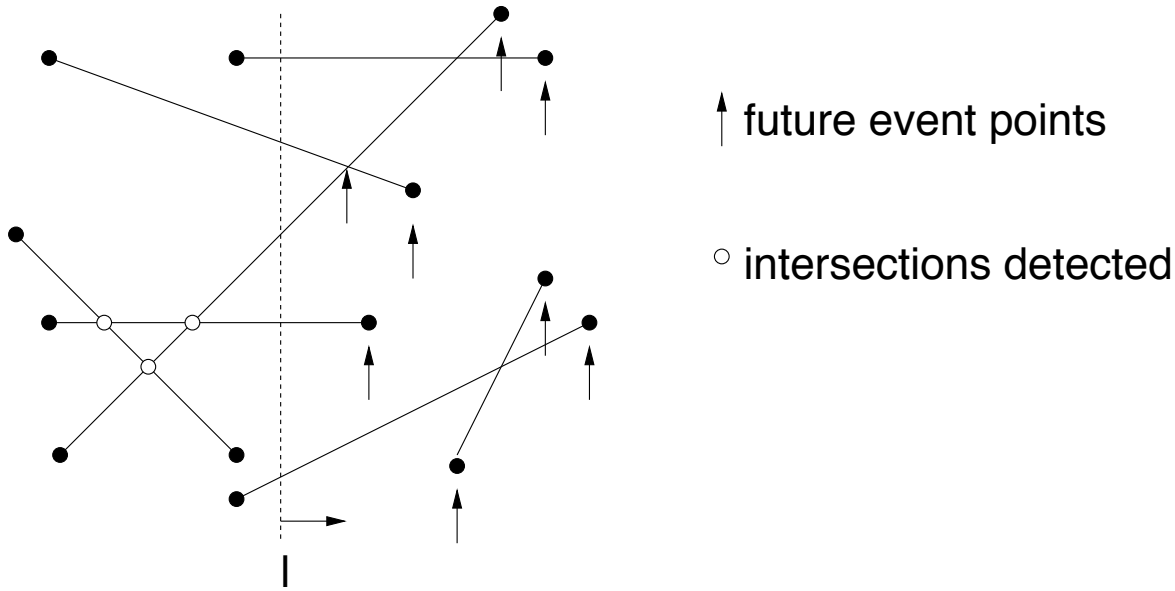
Intersection of line segments



Different applications of plane sweep will have different notions of what event points are. For this application, event points will correspond to the following:

Endpoint events: where the sweep line encounters an endpoint of a line segment, and

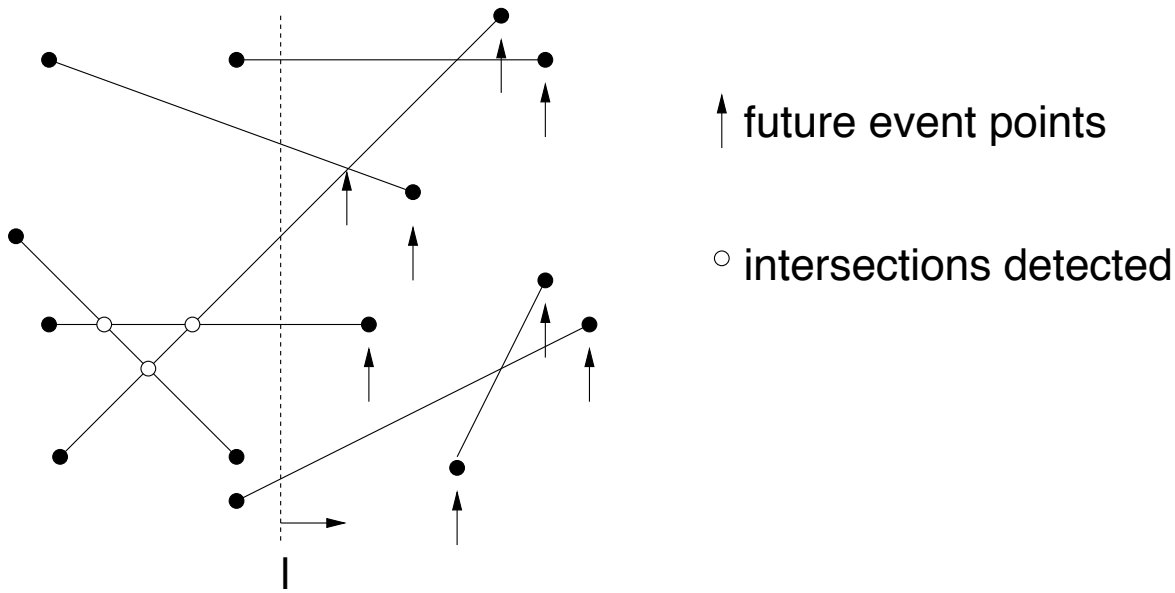
Intersection of line segments



Intersection events: where the sweep line encounters an intersection point of two line segments.

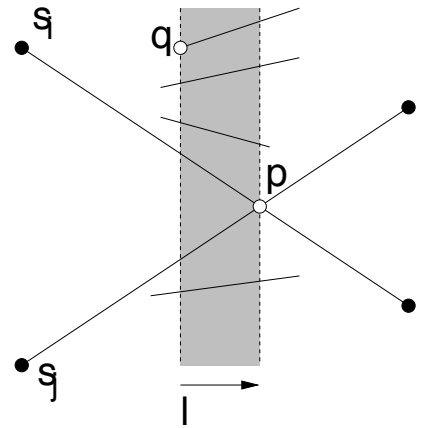
Note that endpoint events can be presorted before the sweep runs. In contrast, intersection events will be discovered as the sweep executes. For example, in the figure, some of the intersection points lying to the right of the sweep line have not yet been "discovered" as events. However, we will show that every intersection point will be discovered as an event before the sweep line arrives here.

Intersection of line segments



Event updates: When an event is encountered, we must update the data structures associated with the event. It is a good idea to be careful in specifying exactly what invariants you intend to maintain. For example, when we encounter an intersection point, we must interchange the order of the intersecting line segments along the sweep line.

Intersection of line segments



A natural question is whether this is sufficient. In particular, if two line segments do intersect, is there necessarily some prior placement of the sweep line such that they are adjacent. Happily, this is the case, but it requires a proof.

Theorem: Given two segments si and sj , which intersect in a single point p (and assuming no other line segment passes through this point) there is a placement of the sweep line prior to this event, such that si and sj are adjacent along the sweep line (and hence will be tested for intersection).

Intersection of line segments

```
Initialize event queue EQ = all segment endpoints;
Sort EQ by increasing x and y;
Initialize sweep line SL to be empty;
Initialize output intersection list IL to be empty;

While (EQ is nonempty) {
    Let E = the next event from EQ;
    If (E is a left endpoint) {
        Let segE = E's segment;
        Add segE to SL;
        Let segA = the segment Above segE in SL;
        Let segB = the segment Below segE in SL;
        If (I = Intersect( segE with segA) exists)
            Insert I into EQ;
        If (I = Intersect( segE with segB) exists)
            Insert I into EQ;
    }
```

```
Else If (E is a right endpoint) {
    Let segE = E's segment;
    Let segA = the segment Above segE in SL;
    Let segB = the segment Below segE in SL;
    Delete segE from SL;
    If (I = Intersect( segA with segB) exists)
        If (I is not in EQ already)
            Insert I into EQ;
}
Else {  // E is an intersection event
    Add E's intersect point to the output list IL;
    Let segE1 above segE2 be E's intersecting segments in SL;
    Swap their positions so that segE2 is now above segE1;
    Let segA = the segment above segE2 in SL;
    Let segB = the segment below segE1 in SL;
    If (I = Intersect(segE2 with segA) exists)
        If (I is not in EQ already)
            Insert I into EQ;
    If (I = Intersect(segE1 with segB) exists)
```

```
            If (I is not in EQ already)
                Insert I into EQ;
    }
    remove E from EQ;
}
return IL;
```

Fortune's Algorithm

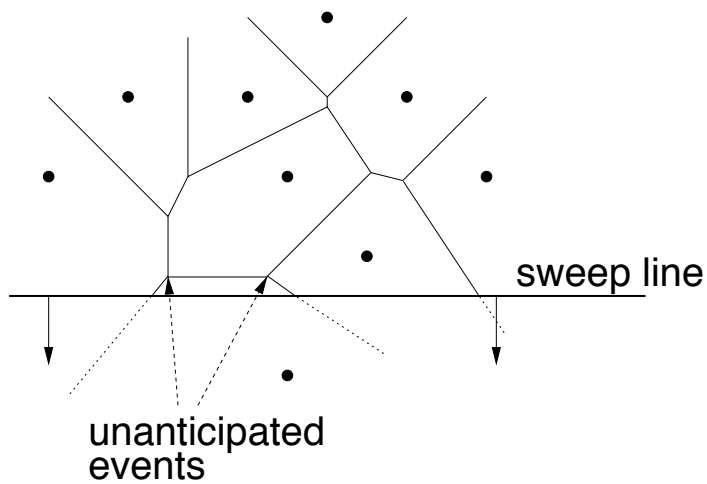
Plane sweep line strategy: a sweep horizontal line going from top to bottom.

Some information is maintained about the intersection of the structure with the sweep line.

While the sweep line l moves downwards the information does not change, except at certain special points: the event points.

Application to compute Voronoi diagrams : Fortune's algorithm.

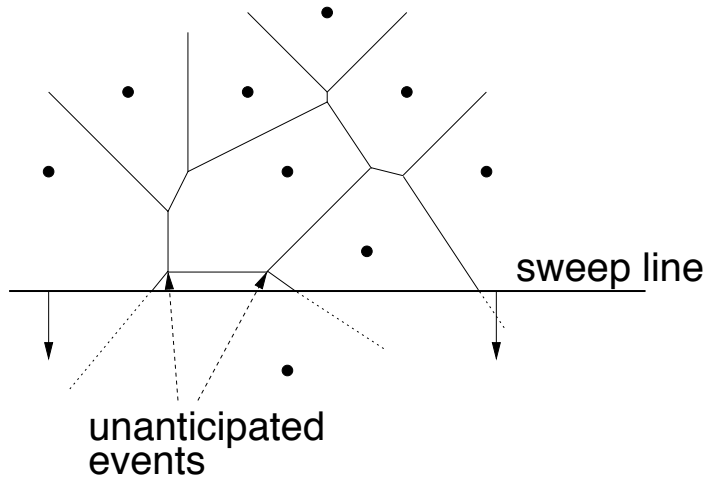
Fortune's Algorithm



Difficulty: the part of Vor(P) above l depends not only on the sites that lie above l but also on sites below l.

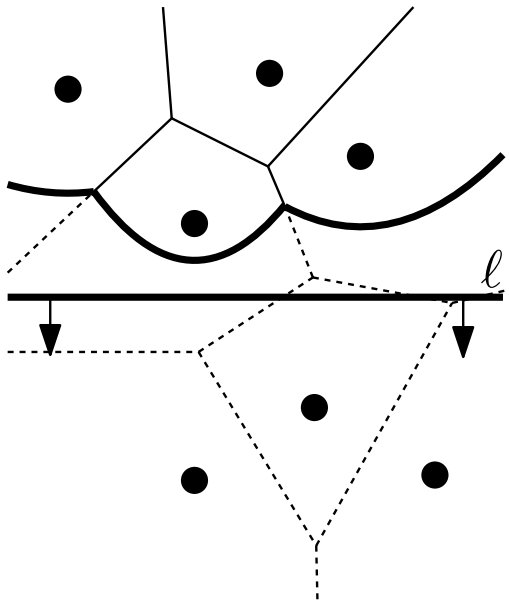
Slight change of strategy: instead of maintaining the intersection of the Voronoi diagram with the sweep line, we maintain information about the part of the Voronoi diagram of the sites above l that cannot be changed by sites below l.

Fortune's Algorithm



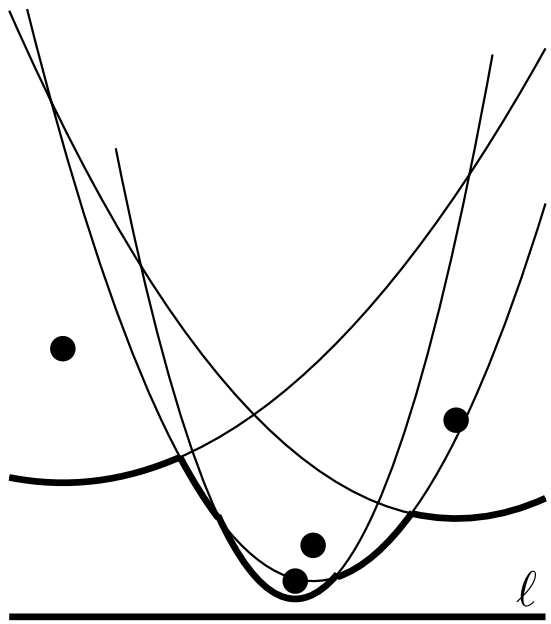
Denote the closed half-plane above l by l^+. What is the part of the Voronoi diagram above l that cannot be changed anymore? In other words, for which points $\mathbf{q} \in l^+$ do we know for sure what their nearest site is? The distance of a point $\mathbf{q} \in l^+$ to any site below l is greater than the distance of $\mathbf{q}$ to l itself. Hence, the nearest site of $\mathbf{q}$ cannot lie below l if $\mathbf{q}$ is at least as near to some site $\mathbf{p}_i \in l^+$ as $\mathbf{q}$ is to l.

Fortune's Algorithm



The locus of points that are closer to some site $\mathbf{p}_i \in l^+$ than to l is bounded by a parabola. Hence, the locus of points that are closer to any site above l than to l itself is bounded by parabolic arcs. We call this sequence of parabolic arcs the beach line.

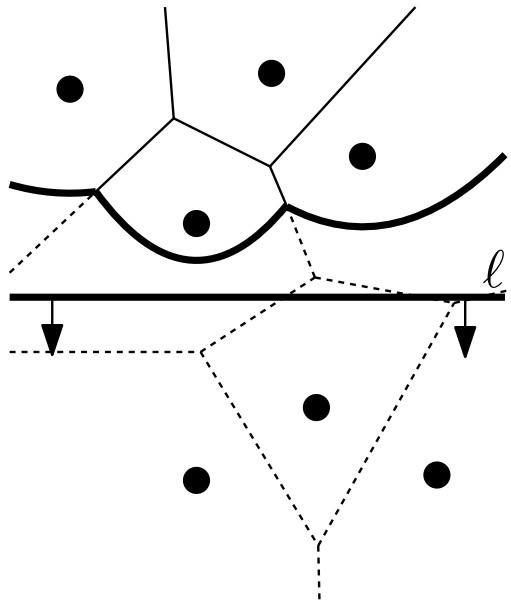
Fortune's Algorithm



One parabola can contribute more than once to the beach line. We'll worry later about how many pieces there can be.

The breakpoints between the different parabolic arcs forming the beach line lie on edges of the Voronoi diagram. This is not a coincidence: the breakpoints exactly trace out the Voronoi diagram while the sweep line moves from top to bottom. These properties of the beach line can be proved using elementary geometric arguments.

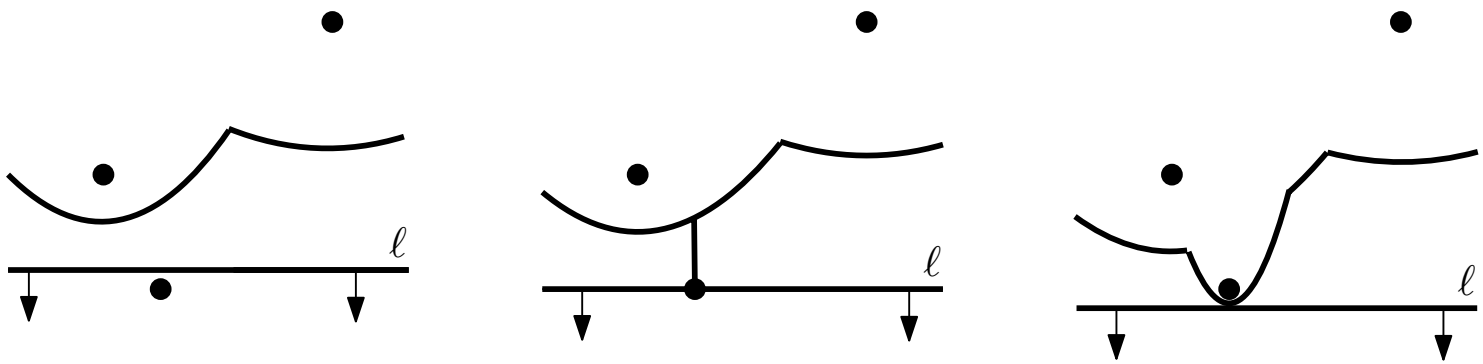
Fortune's Algorithm



Instead of maintaining the intersection of Vor(P) with l we maintain the beach line as we move our sweep line l.

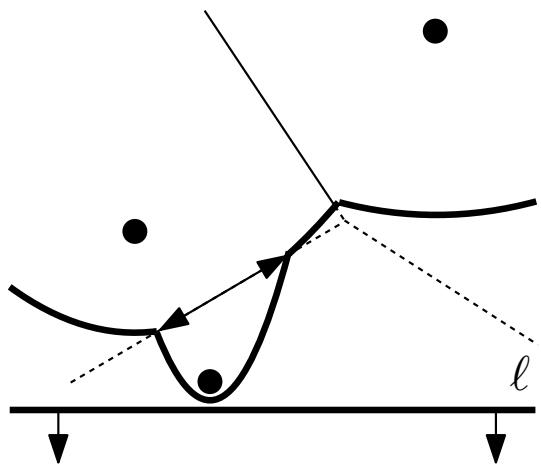
We do not maintain the beach line explicitly, since it changes continuously as l moves. For the moment let's ignore the issue of how to represent the beach line until we understand where and how its combinatorial structure changes. This happens when a new parabolic arc appears on it, and when a parabolic arc shrinks to a point and disappears.

Fortune's Algorithm



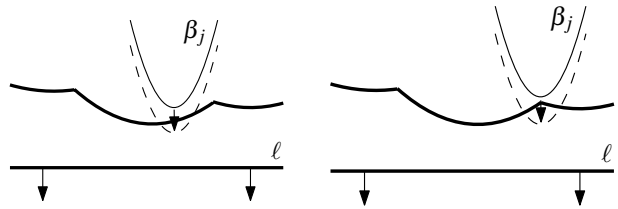
First we consider the events where a new arc appears on the beach line. One occasion where this happens is when the sweep line l reaches a new site. The parabola defined by this site is at first a degenerate parabola with zero width: a vertical line segment connecting the new site to the beach line. As the sweep line continues to move downward the new parabola gets wider and wider. The part of the new parabola below the old beach line is now a part of the new beach line. We call the event where a new site is encountered a site event.

What happens to $\mathrm{Vor}(P)$ at a site event?



Breakpoints on the beach line trace out the edges of the Voronoi diagram. At a site event two new breakpoints appear, which start tracing out edges. In fact, the new breakpoints coincide at first, and then move in opposite directions to trace out the same edge. Initially, this edge is not connected to the rest of the Voronoi diagram above the sweep line. Later on,the growing edge will run into another edge, and it becomes connected to the rest of the diagram.

What happens to $\mathrm{Vor}(P)$ at a site event?



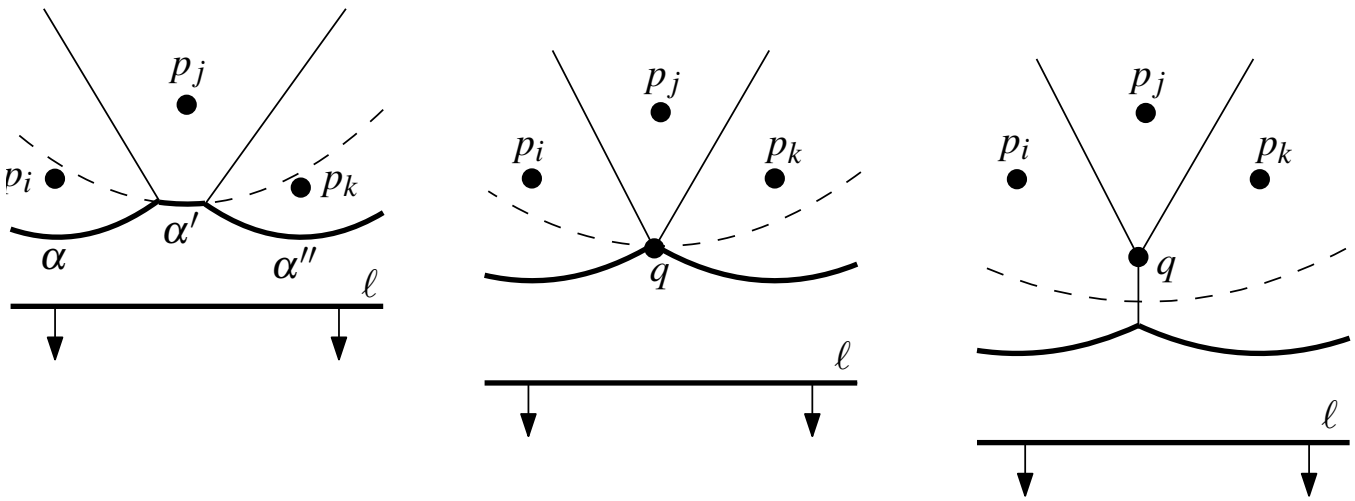
So now we understand what happens at a site event: a new arc appears on the beach line, and a new edge of the Voronoi diagram starts to be traced out.

Is it possible that a new arc appears on the beach line in any other way? The answer is no.

Theorem : The only way in which a new arc can appear on the beach line is through a site event.

Demonstration is skipped. An immediate consequence of the lemma is that the beach line consists of at most $2N - 1$ parabolic arcs: each site encountered gives rise to one new arc and the splitting of at most one existing arc into two, and there is no other way an arc can appear on the beach line.

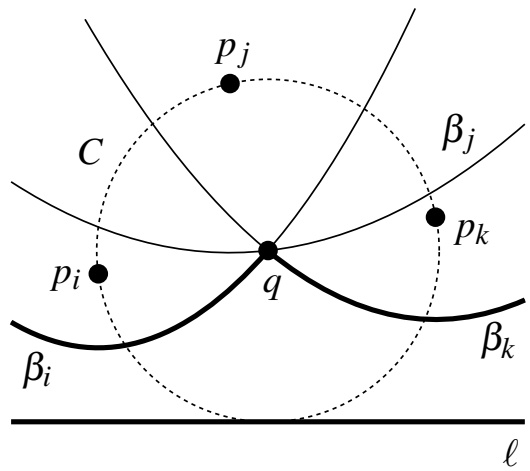
Fortune's algorithm



The second type of event in the plane sweep algorithm is where an existing arc of the beach line shrinks to a point and disappears.

The arcs α and α'' cannot be part of the same parabola. Hence, the three arcs α, α', and α'' are defined by three distinct sites $\mathbf{p}_i$, $\mathbf{p}_j$, and $\mathbf{p}_k$. At the moment α' disappears, the parabolas defined by these three sites pass through a common point $\mathbf{q}$. Point $\mathbf{q}$ is equidistant from l and each of the three sites.

Fortune's algorithm



When an arc disappears from the beach line and two breakpoints meet, two edges of the Voronoi diagram meet as well. We call the event where the sweep line reaches the lowest point of a circle through three sites defining consecutive arcs on the beach line a circle event.

Theorem : The only way in which an existing arc can disappear from the beach line is through a circle event.

Combinatorial structure of the beach line

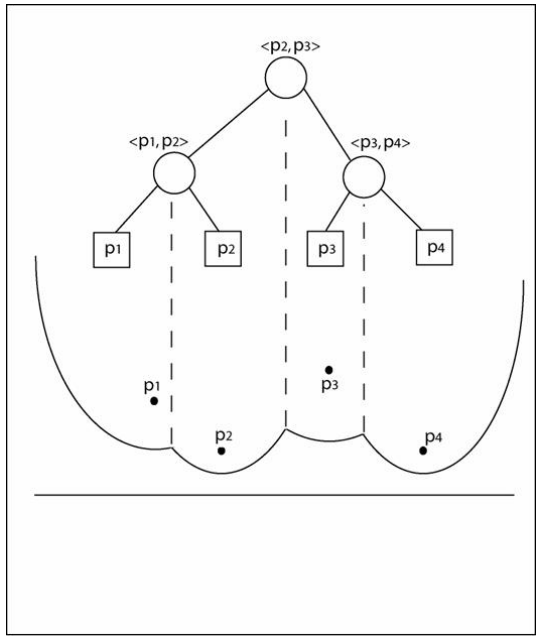
We know where and how the combinatorial structure of the beach line changes: at a site event a new arc appears, and at a circle event an existing arc drops out.

We also know how this relates to the Voronoi diagram under construction: at a site event a new edge starts to grow, and at a circle event two growing edges meet to form a vertex.

It remains to find the right data structures to maintain the necessary information during the sweep. Our goal is to compute the Voronoi diagram, so we need a data structure that stores the part of the Voronoi diagram computed thus far. We also need the two standard data structures for any sweep line algorithm: an event queue and a structure that represents the status of the sweep line.

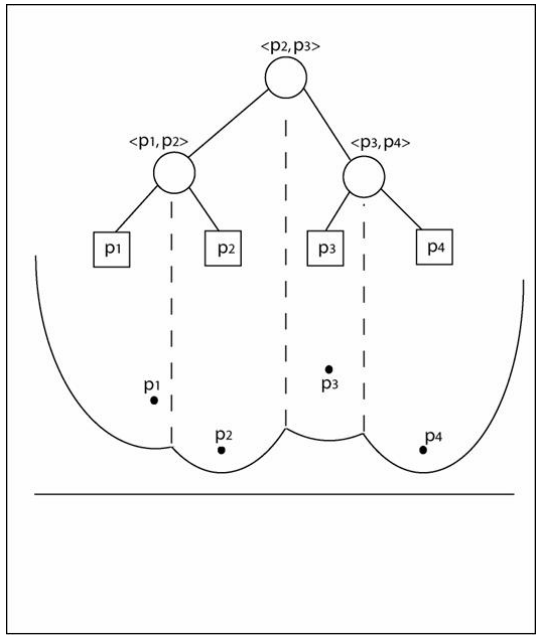
Sweep-line algorithm

Beach line: The beach line is represented using a dictionary e.g. a balanced binary tree $\mathcal{T}$. It is the status structure. Its leaves correspond to the arcs of the beach line, which is x-monotone. Each leaf μ stores the site that defines the arc it represents. The internal nodes of $\mathcal{T}$ represent the breakpoints of the beach line.



Notice that a site may appear multiple times on the beach line

Fortune's algorithm



Algorithm

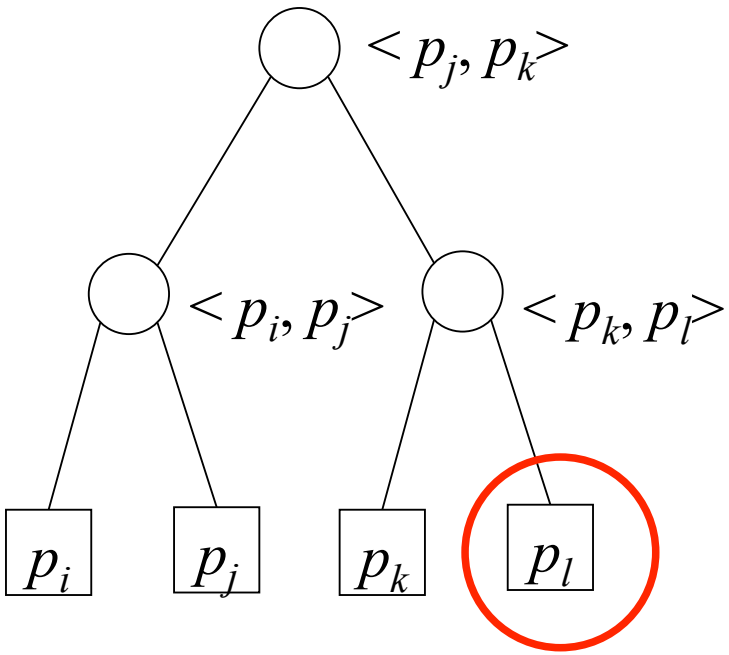
1. Initialize
   - Event queue Q ← all site events
   - Binary search tree T ← ∅
   - Doubly linked list D ← ∅
2. While Q not ∅,
   - Remove event (e) from Q with largest y-coordinate
     - HandleEvent(e, T, D)

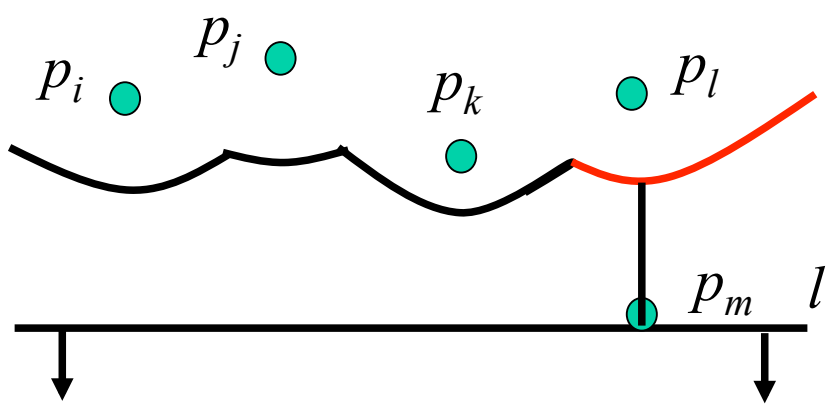
Handling Site Events

1. Locate the existing arc (if any) that is above the new site
2. Break the arc by replacing the leaf node with a sub tree representing the new arc and its break points
3. Add two half-edge records in the doubly linked list
4. Check for potential circle event(s), add them to event queue if they exist

Locate the existing arc that is above the new site

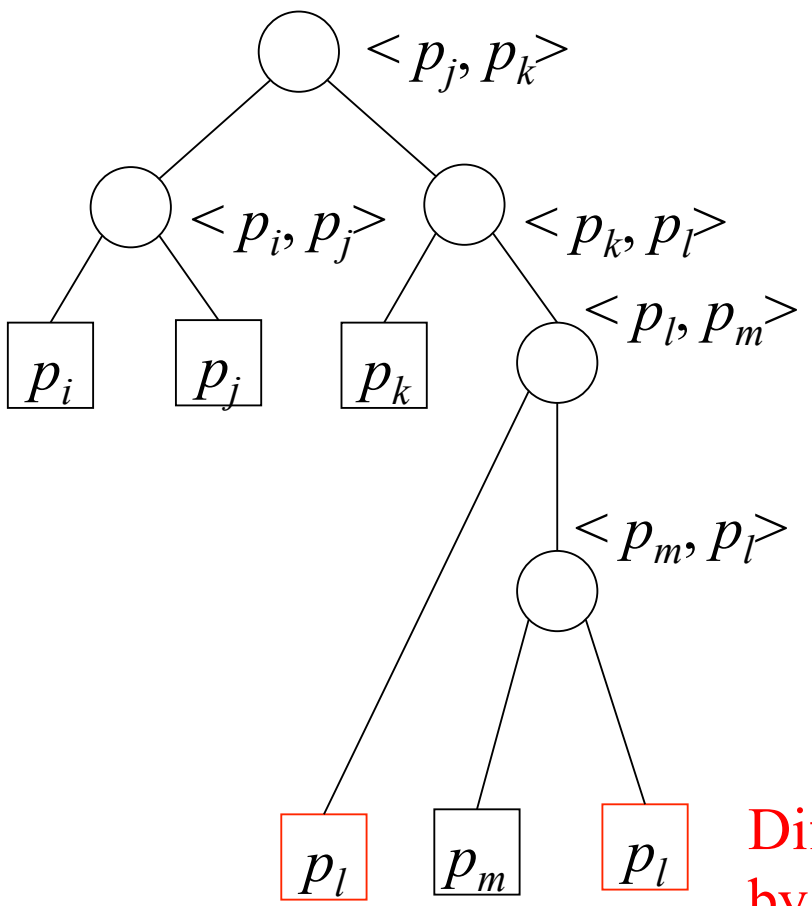
- The x coordinate of the new site is used for the binary search
- The x coordinate of each breakpoint along the root to leaf path is computed on the fly

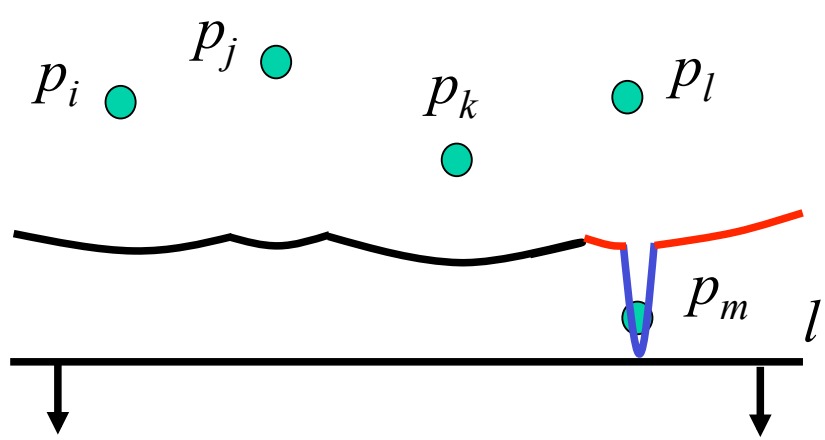




Break the Arc

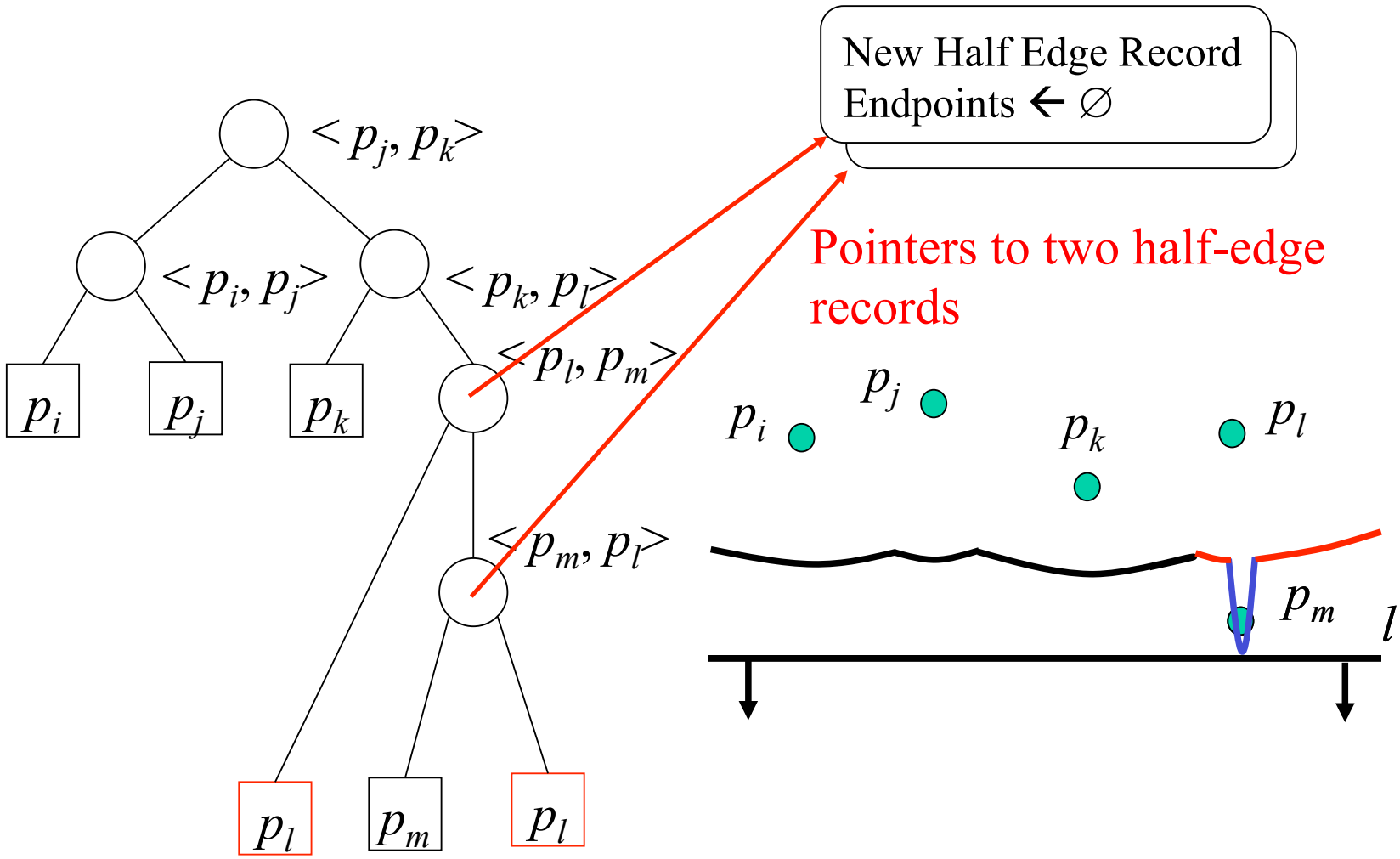
Corresponding leaf replaced by a new sub-tree





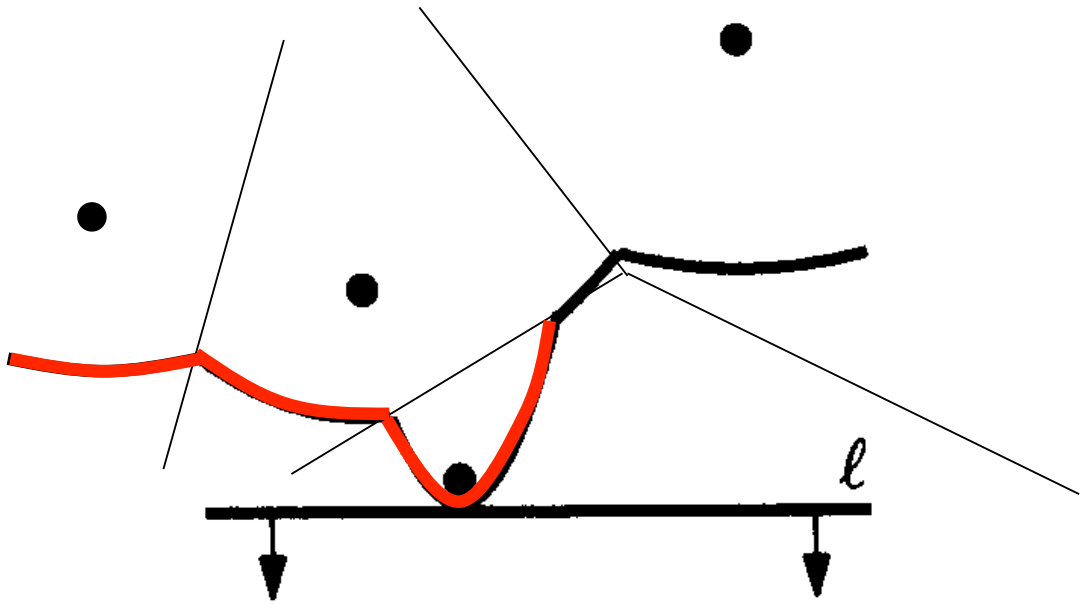
Different arcs can be identified by the same site!

Add a new edge record in the doubly linked list



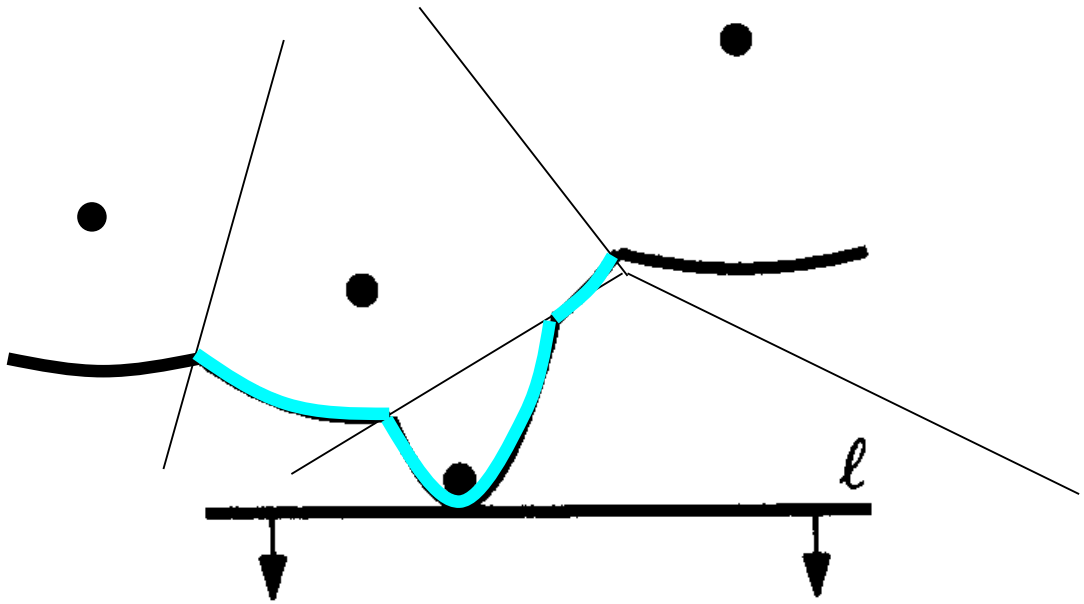
Checking for Potential Circle Events

- Scan for triple of consecutive arcs and determine if breakpoints converge
  - Triples with new arc in the middle do not have break points that converge



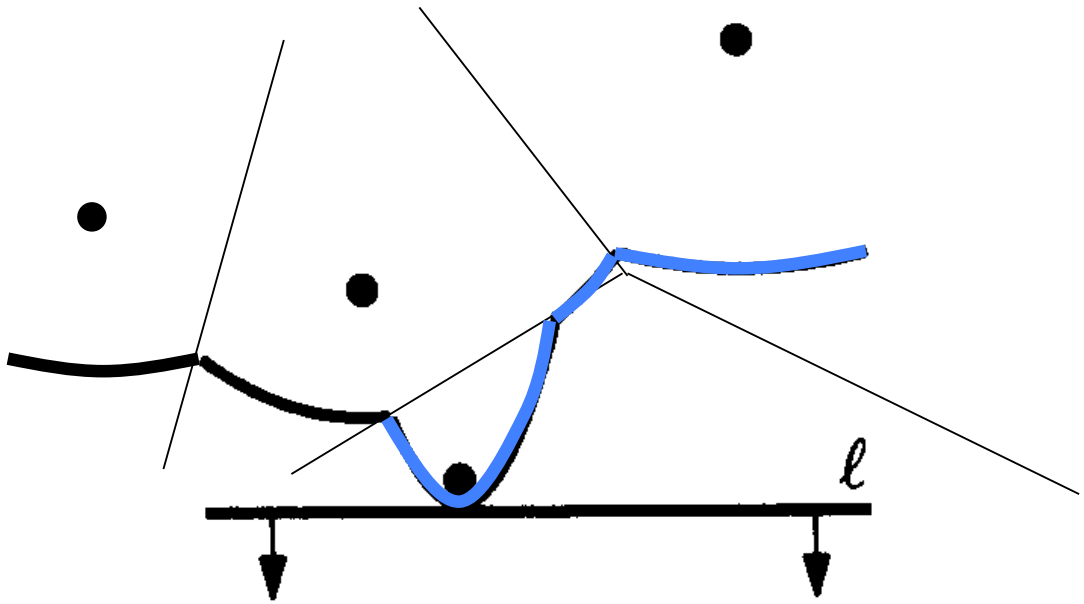
Checking for Potential Circle Events

- Scan for triple of consecutive arcs and determine if breakpoints converge
  - Triples with new arc in the middle do not have break points that converge



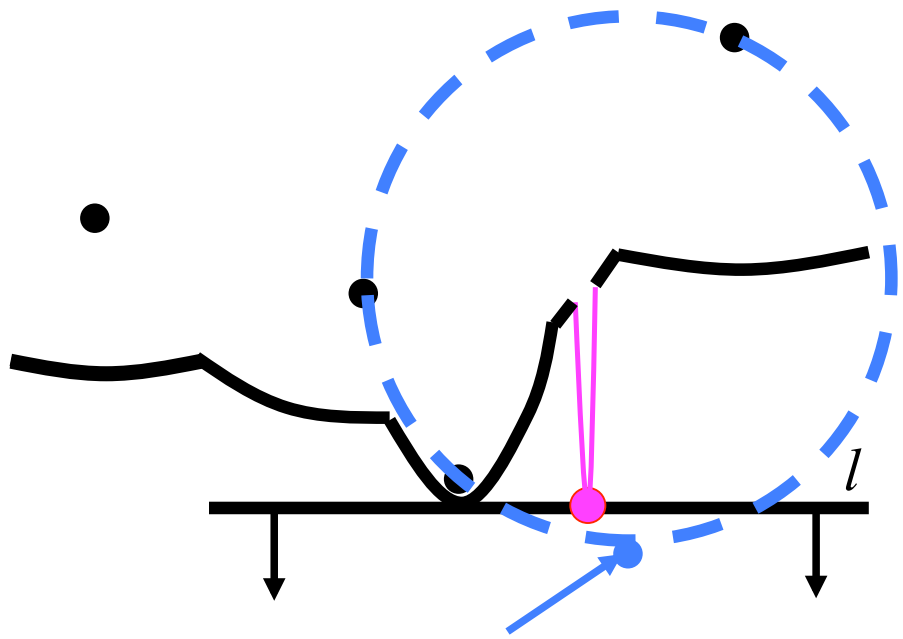
Checking for Potential Circle Events

- Scan for triple of consecutive arcs and determine if breakpoints converge
  - Triples with new arc in the middle do not have break points that converge



Converging break points may not always yield a circle event

- Appearance of a new site before the circle event makes the potential circle non-empty



(The original circle event becomes a *false alarm*)

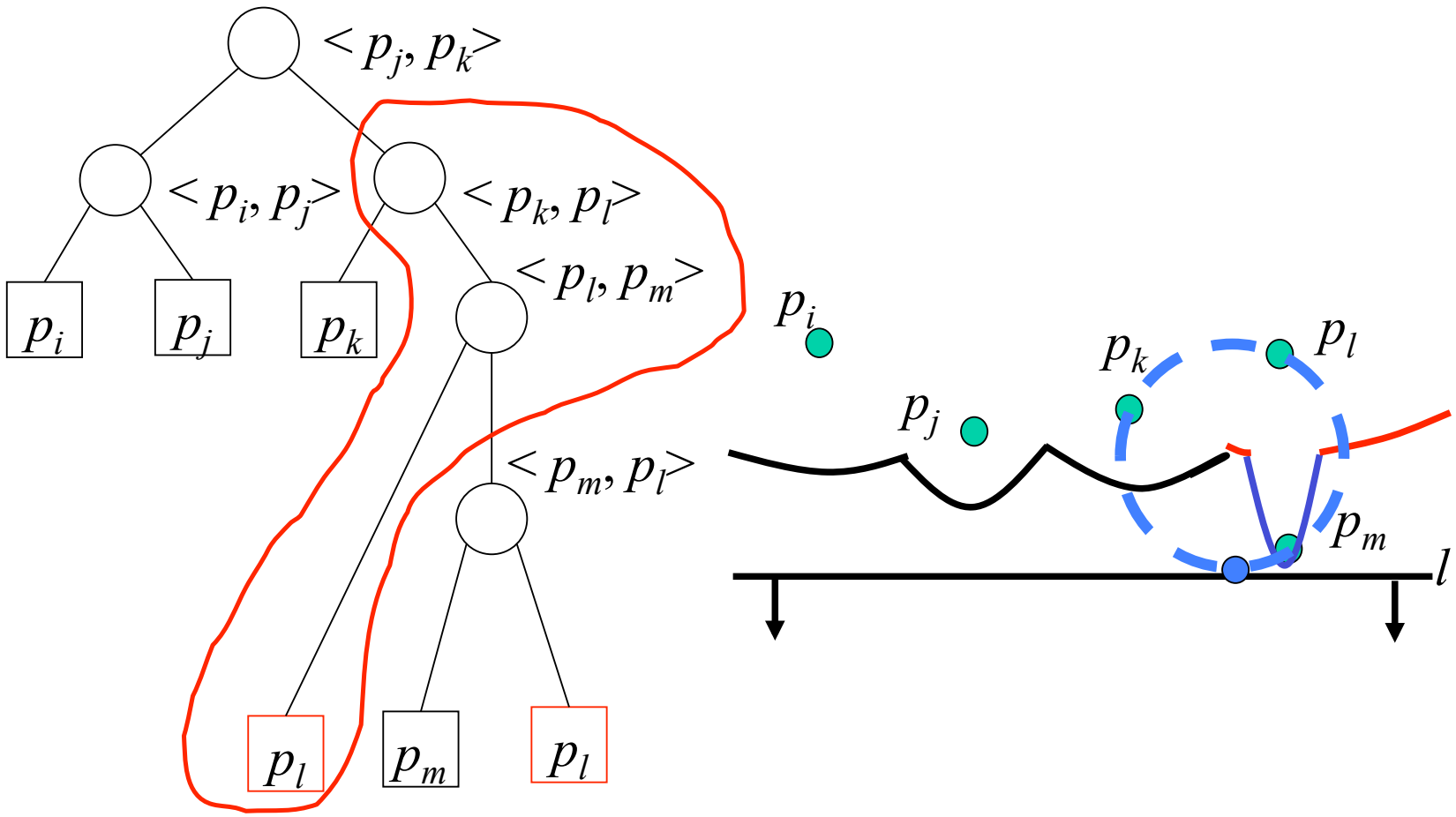
Handling Site Events

1. Locate the leaf representing the existing arc that is above the new site
   - Delete the potential circle event in the event queue
2. Break the arc by replacing the leaf node with a sub tree representing the new arc and break points
3. Add a new edge record in the doubly linked list
4. Check for potential circle event(s), add them to queue if they exist
   - Store in the corresponding leaf of T a pointer to the new circle event in the queue

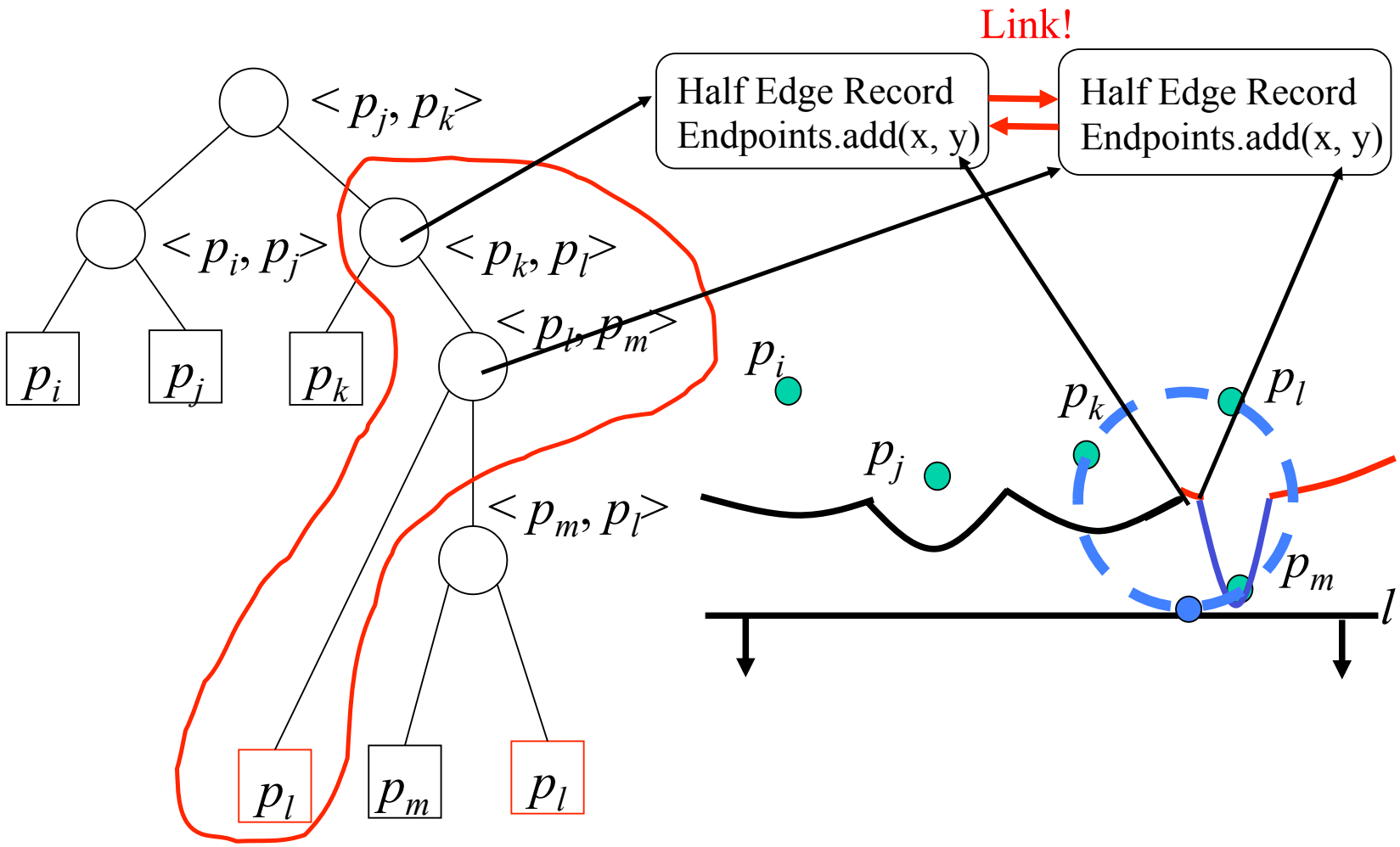
Handling Circle Events

1. Add vertex to corresponding edge record in doubly linked list
2. Delete from T the leaf node of the disappearing arc and its associated circle events in the event queue
3. Create new edge record in doubly linked list
4. Check the new triplets formed by the former neighboring arcs for potential circle events

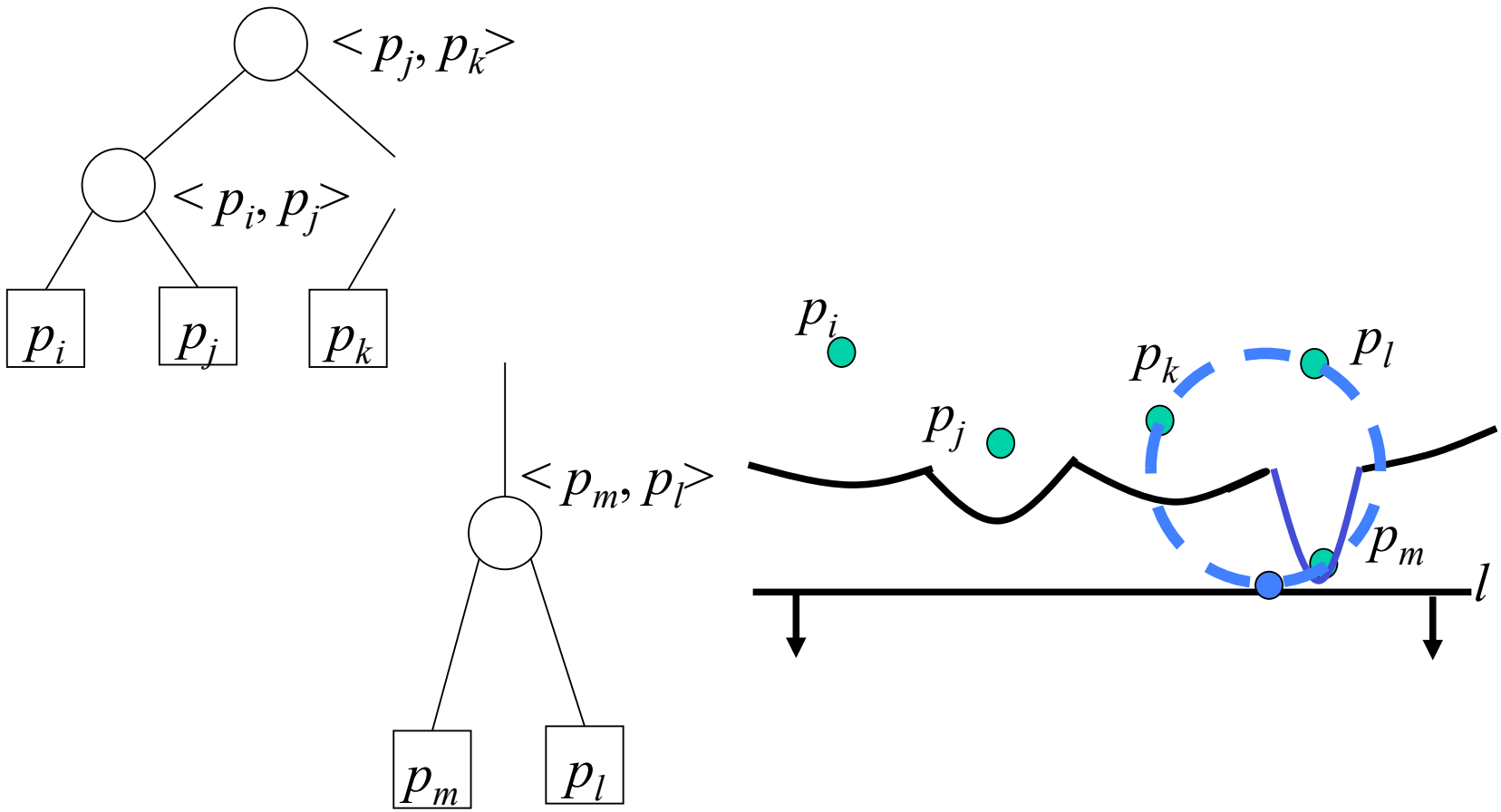
A Circle Event



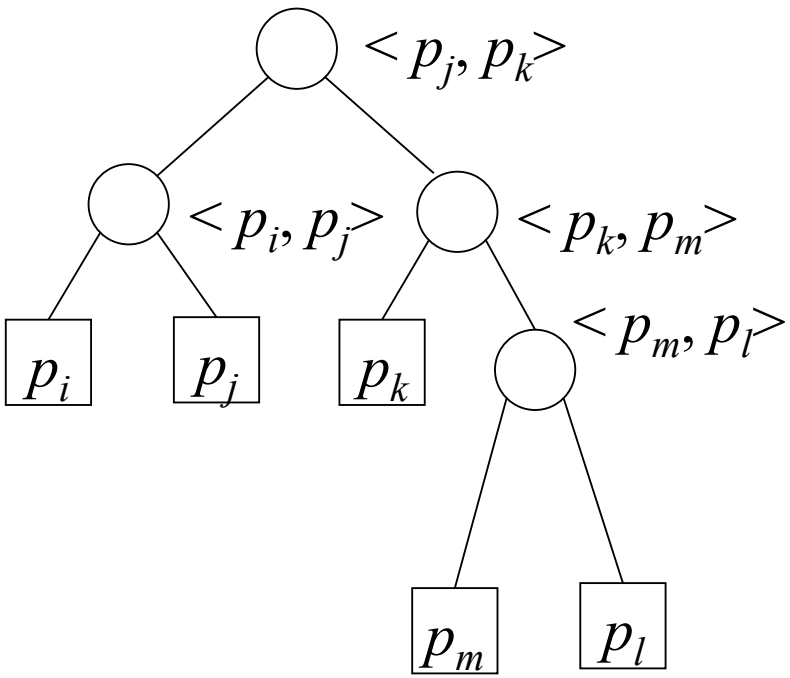
Add vertex to corresponding edge record

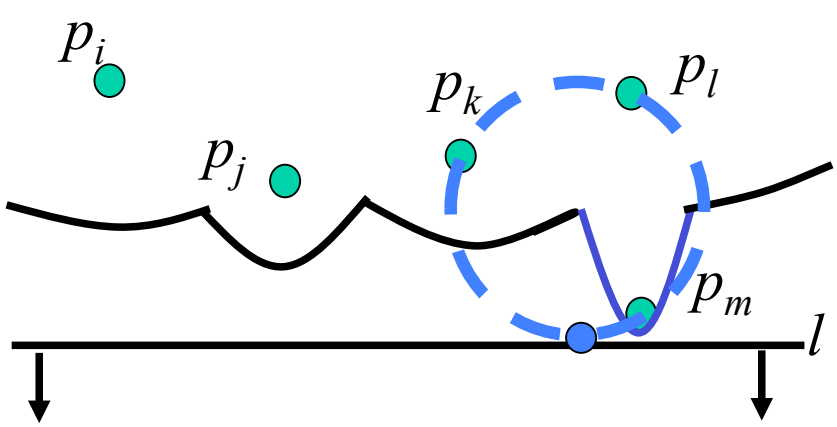


Deleting disappearing arc

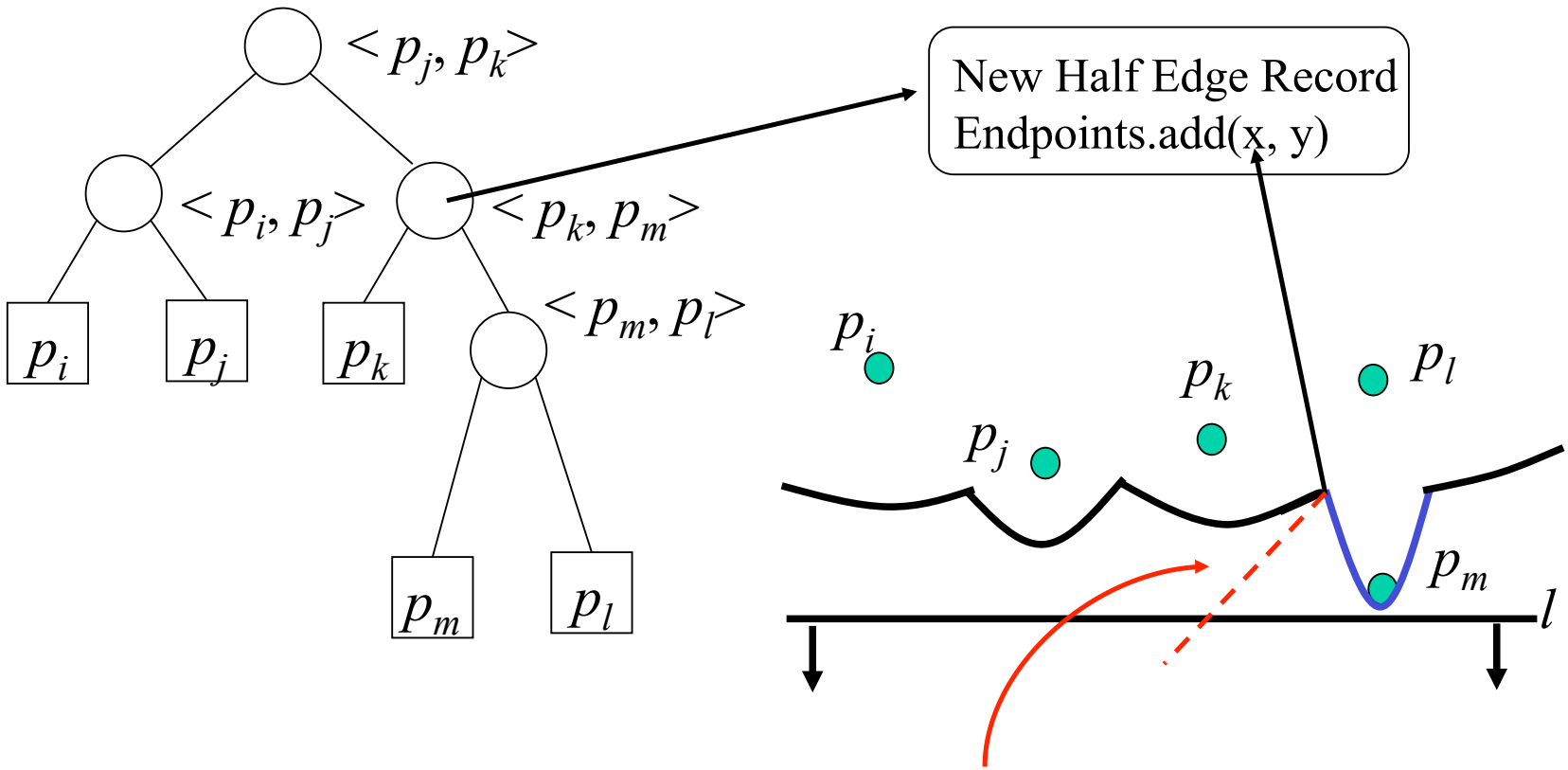


Deleting disappearing arc



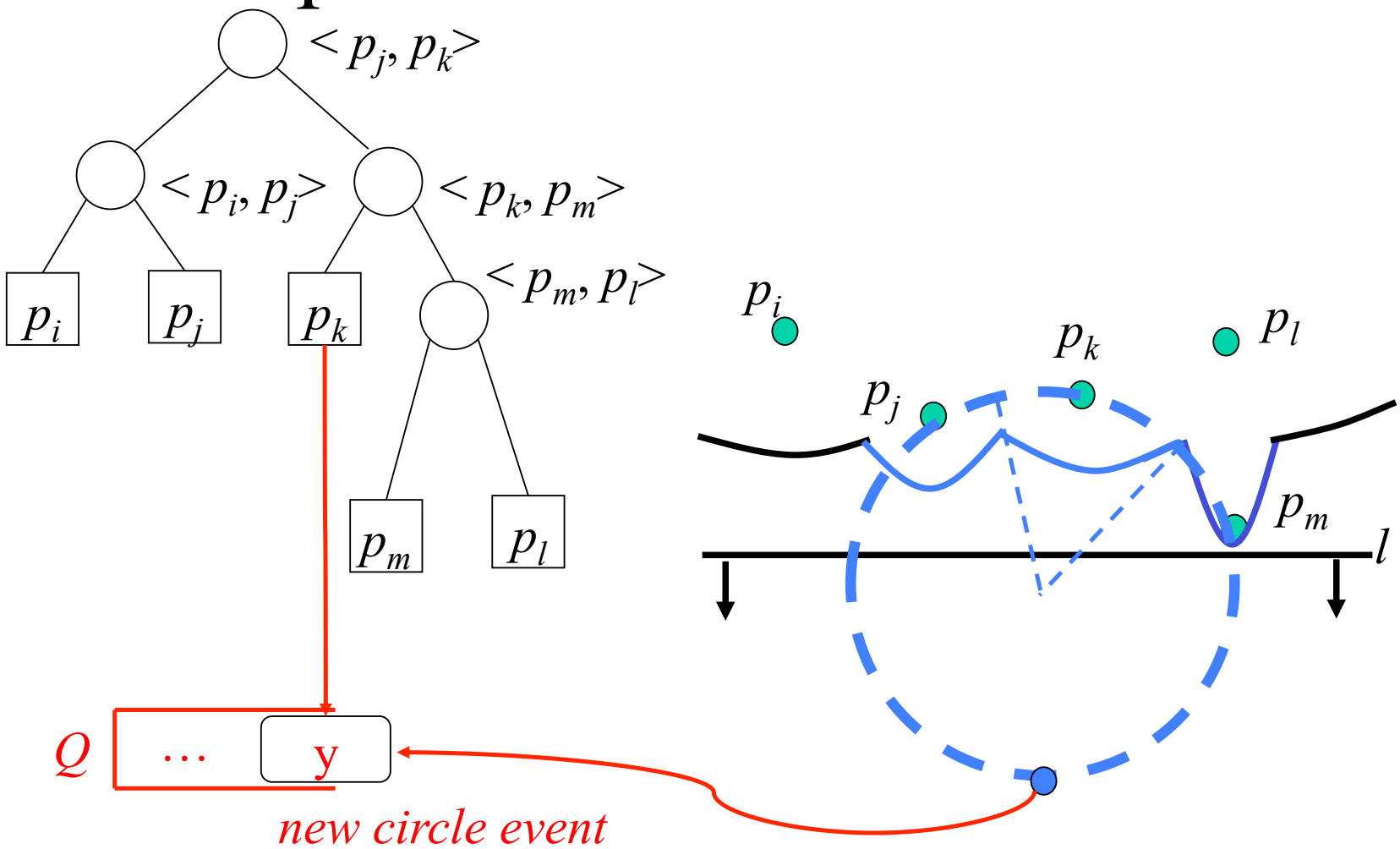


Create new edge record



A new edge is traced out by the new break point $<p_k, p_m>$

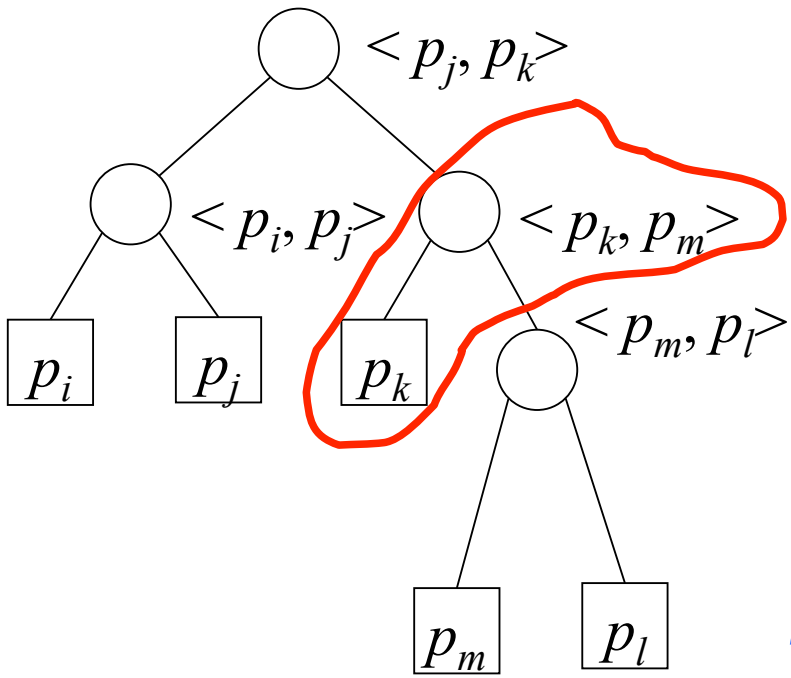
Check the new triplets for potential circle events

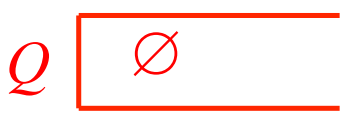


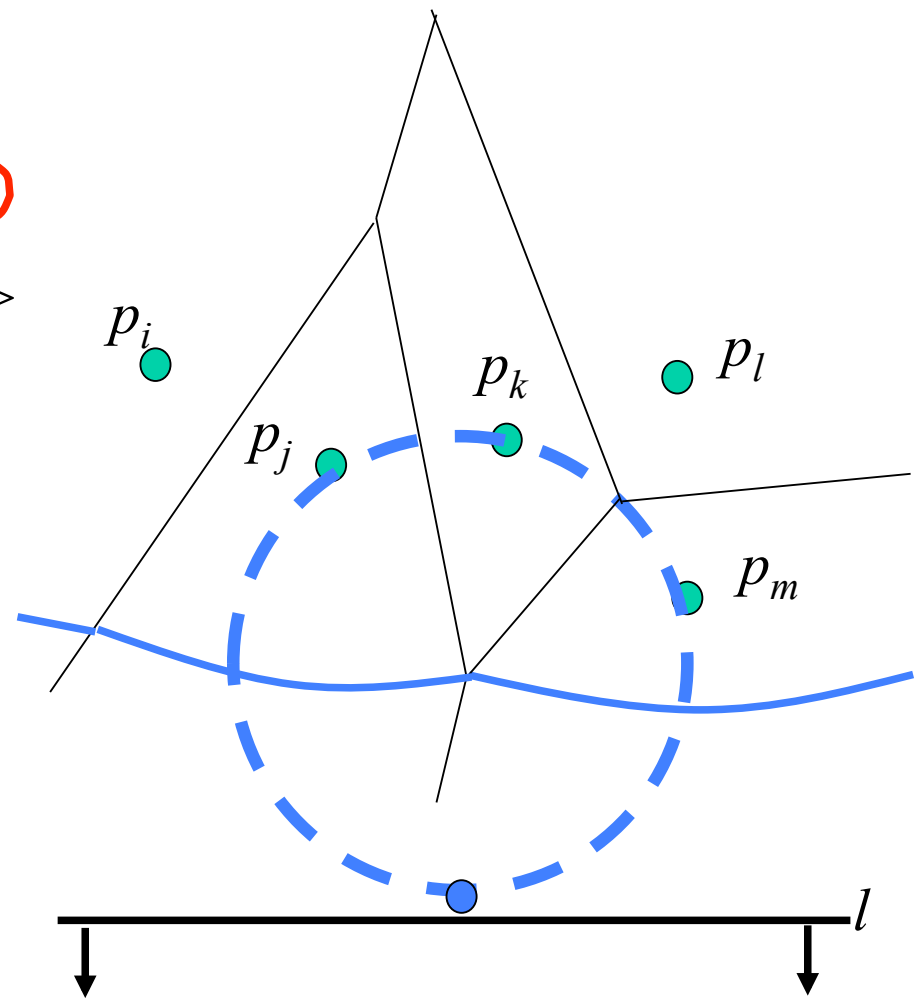
Minor Detail

- Algorithm terminates when Q = ∅, but the beach line and its break points continue to trace the Voronoi edges
  - Terminate these "half-infinite" edges via a bounding box

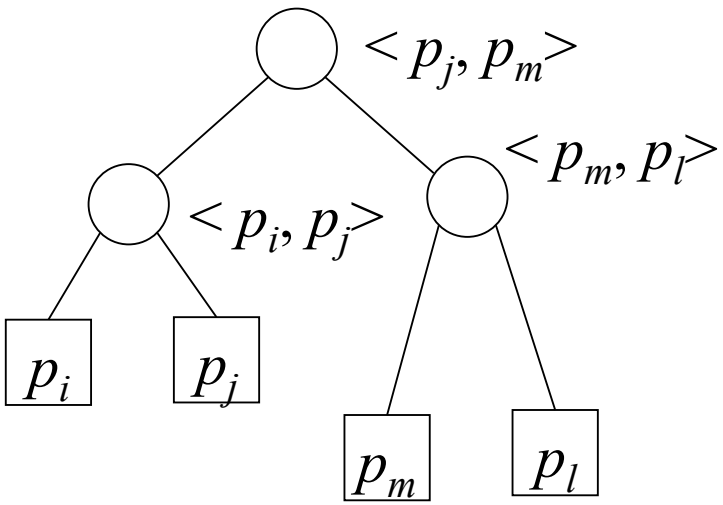
Algorithm Termination

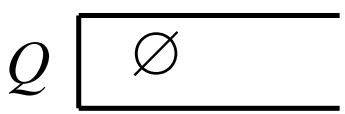


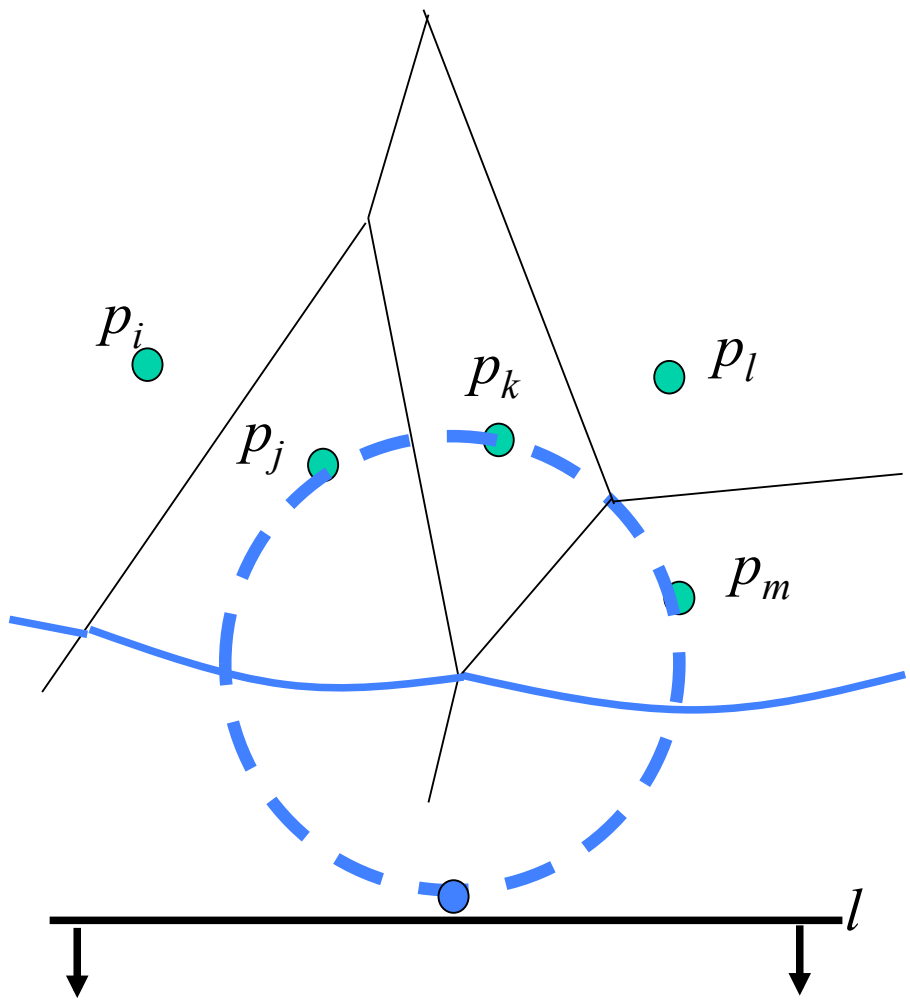




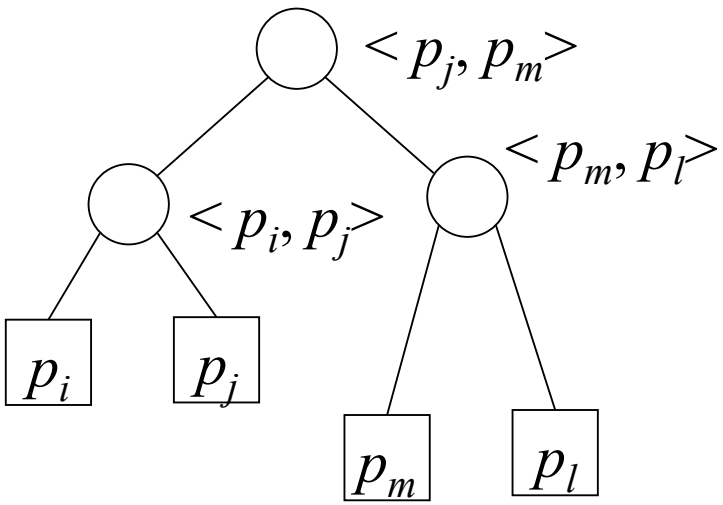
Algorithm Termination

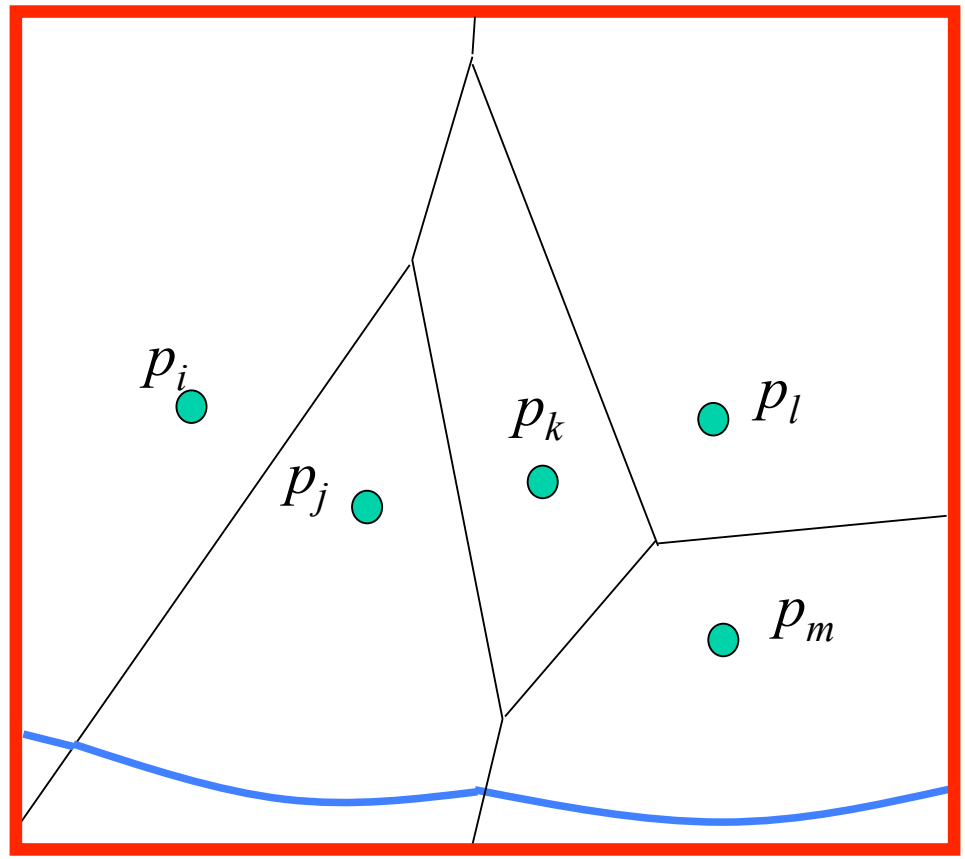




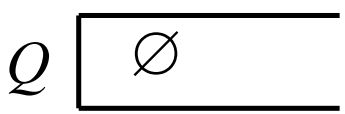


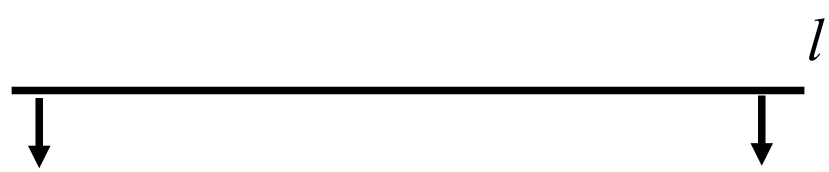
Algorithm Termination





Terminate half-lines with a bounding box!





Handling Site Events

	Running Time
1. Locate the leaf representing the existing arc that is above the new site	$O(\log n)$
– Delete the potential circle event in the event queue	
2. Break the arc by replacing the leaf node with a sub tree representing the new arc and break points	$O(1)$
3. Add a new edge record in the link list	$O(1)$
4. Check for potential circle event(s), add them to queue if they exist	$O(1)$
– Store in the corresponding leaf of T a pointer to the new circle event in the queue	

Handling Circle Events

	Running Time
1. Delete from T the leaf node of the disappearing arc and its associated circle events in the event queue	$O(\log n)$
2. Add vertex record in doubly link list	$O(1)$
3. Create new edge record in doubly link list	$O(1)$
4. Check the new triplets formed by the former neighboring arcs for potential circle events	$O(1)$

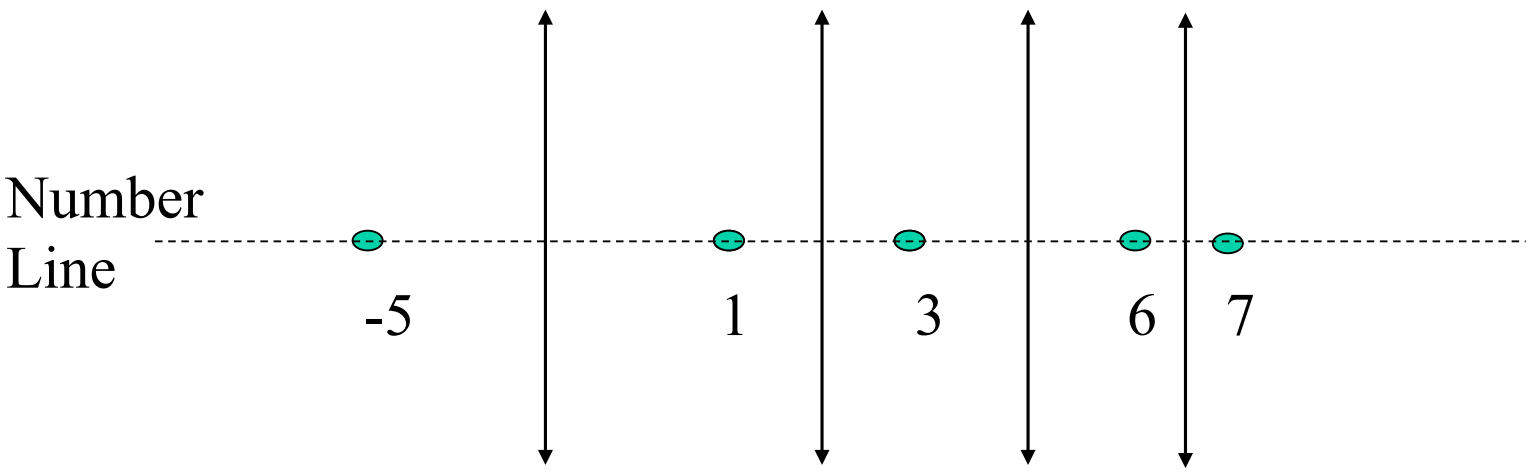
Total Running Time

- Each new site can generate at most two new arcs
  - →beach line can have at most $2n - 1$ arcs
  - →at most $O(n)$ site and circle events in the queue
- Site/Circle Event Handler $O(\log n)$

→ $O(n \log n)$ total running time

Is Fortune's Algorithm Optimal?

- We can sort numbers using any algorithm that constructs a Voronoi diagram!



- Map input numbers to a position on the number line. The resulting Voronoi diagram is doubly linked list that forms a chain of unbounded cells in the left-to-right (sorted) order.

Summary

- Voronoi diagram is a useful planar subdivision of a discrete point set
- Voronoi diagrams have linear complexity and can be constructed in $O(n \log n)$ time
- Fortune's algorithm (optimal)

.