

Planar Point Location Problem with Persistent Search Trees

James Nhan (160149990)

April 6, 2017

1 Introduction

The point location problem described by Neil Sarnak and Robert E. Tarjan in their paper "Planar point location using persistent search trees" [?] is one that takes a planar subdivision and a point. The goal is to determine which polygon the point resides in. Sarnak and Tarjan also detail using a specific type of data structure that is persistent across insertions and deletions. This enables a very fast query time while keeping size down. The associated program implements the solution described. A link to the GitHub repository can be found in at the end of the report.

2 Program Description

The program implements three main steps: constructing a doubly connected edge list (DCEL) from a simple polygon, monotone and triangulate a simple polygon, and creating a `PointLocation` object using persistent red black trees. From that point, the user can then query the point location object with a `Point` object. This query will return the a half-edge that bounds the face containing the point. The program utilizes the sweep line method of creating a triangulated monotone polygon and the slab method to perform the point location query. This program utilizes an open source implementation of a persistent red black tree set by Tom Larkworthy [?] as described by Sarnak and Tarjan [?].

3 Features and Limitations

3.1 Features

The first main feature of the program is the implementation of a DCEL for describing the polygons. Begin with an array of `Point` objects. Each of which should represent a point of the polygon's outer boundary in counter-clockwise order. For some DCEL object, invoke its member method `ConstructSimplePolygon` method, passing in the array of `Point` objects. The DCEL object will contain the description of the polygon as a `HashSet` of `Face`, `HalfEdge`, and `Vertex` objects.

Now, with a DCEL, we can monotone it through its member method `MakeMonotone` and triangulate the resulting monotone polygon through another member method `TriangulateMonotonePolygon`.

Finally, we may construct a `PointLocation` object. The constructor will take in a DCEL and construct a persistent red black tree set to contain slabs. This `PointLocation` object can then be used to query the planar subdivision with `Point` objects via the `Query` member method. The method is very fast. It can be done in $O(\log n)$ time. The size of the tree grows in $O(n)$ space due to the persistent nature of the

tree and use of limited node copying. The description that Larkworthy provides mistakenly states that the size is $O(\log n)$.

3.2 Limitations

This program does have several limitations. It will not handle non-simple polygons. That is, any polygons with intersecting edges will fail to construct a simple polygon. In the event that `ConstructSimplePolygon` is invoked on a set of `Point` objects that define a polygon with intersecting points, an exception will be thrown.

Another limitation stems from polygons with holes. Because the program defines the polygons via a list of points in counter-clockwise order, it is not possible to define a polygon with a hole; although, legacy code lingers to support it throughout. This limitation is confined to the construction of the polygon. Other parts of the program are not subject to this limitation with the exception of monotonizing and triangulating in the case of an edge going through a hole.

Adding a `Vertex` to the polygon only works when adding one directly on an edge to split the edge; however, adding in the capability to add a vertex anywhere would not be difficult. We would simply have to get the two nearest vertices, add edges from them to the new `Vertex`, and finally, delete the edge between the two vertices.

The polygon does contain a reference to the `Face` containing the polygon. That is, the `Face` that is unbounded on the outside. Should the point location query receive a `Point` that is within this `Face`, the query result will be `null`. If the user would like to instead retrieve a reference to the `Face` or `HalfEdge` bounding this face on the inside, a modification can be made in the `Query` method. Instead of returning `null` at the end, loop through the faces of the polygon and return the `Face` whose `GetInner()` object is `null`. This is the `Face` unbounded on the outside.

To summarize the limitations, this program can only handle simple polygons that have no holes.

4 Compiling and Running

The original development environment of this project was in Eclipse Neon.3 Release (4.613). The build id is 20170314-1500. The JDK version used is 1.8.0_91 and the JRE version targeted is 1.8.0_91. Included in the GitHub repository are the Eclipse project files, and the project should compile and run out of the box.

If the user wishes to compile and run via command line, the following command should be issued:

```
javac -d bin/ src/*.java
```

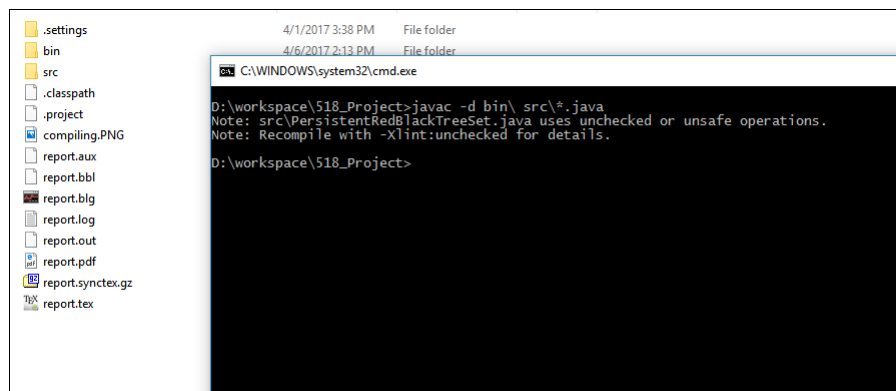


Figure 1: Compiling the project.

To run from the command line, issue the command:

```
java -cp bin/ bin/Main
```

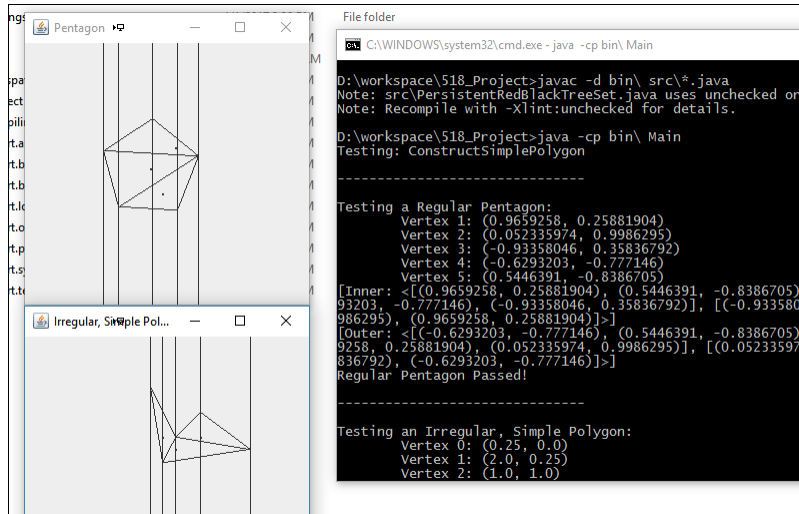


Figure 2: Running the project.

The result will be the test output in the command line and two Swing windows to visualize the polygons, their triangulations, slabs, and points tested in the point location problem.

The latest version of the project can be obtained from the GitHub repository:

https://github.com/jamesnhan/518_Project

5 API

5.1 DCEL

`DCEL()` - The default constructor. It will initialize an empty DCEL object.

`AddEdge(Vertex, Vertex)` - Adds an edge between the two Vertex objects.

`AddVertex(Vertex)` - Adds a Vertex to the polygon. The Vertex must be on an edge of the polygon.

`ConstructSimplePolygon(Point[])` - Constructs a simple polygon from the Point array and stores it in the DCEL. Throws an exception for non-simple polygons.

`GetEdge()` : `HashSet<HalfEdge>` - Returns the HalfEdge objects of the polygon.

`GetFaces()` : `HashSet<Face>` - Returns the Face objects of the polygon.

`GetVertices()` : `HashSet<Vertex>` - Returns the Vertex objects of the polygon.

`IsSimplePolygon(Vertex[])` : `boolean` - Checks whether or not the DCEL is a simple polygon.

`IsSimplePolygon(HashSet<Vertex>)` : `boolean` - Checks whether or not the DCEL is a simple polygon.

5.2 Event

`Event(HalfEdge, Vertex)` - The constructor. It will initialize an `Event` for the `Vertex` and the associated `HalfEdge`.

`compareTo(Event) : int` - Compares the `Event` based on the incident `Vertex`.

`GetEdge() : HalfEdge` - Returns the `HalfEdge` incident on the event point.

`GetType() : EventType` - Gets the type of `Event` in the form of the `EventType` enum.

`GetVertex() : Vertex` - Returns the `Vertex` incident on the event point.

`SetEdge(HalfEdge)` - Sets the incident `HalfEdge`.

`SetType(EventType)` - Sets the type of `Event` in the form of the `EventType` enum.

`SetVertex(Vertex)` - Sets the incident `Vertex`.

5.2.1 EventType

`LEFT` - An `Event` taking place on the left `Vertex` of an edge.

`RIGHT` - An `Event` taking place on the right `Vertex` of an edge.

5.3 EventQueue

`EventQueue(Vertex[])` - The constructor. It will initialize an `EventQueue` for the `Vertex` array given with an `Event` for each `Vertex`.

5.4 Face

`Face()` - The default constructor. It will initialize an unbounded `Face`.

`Face(HalfEdge, HalfEdge)` - The constructor. It will initialize a `Face` bounded on the outside by the first `HalfEdge` and with a single `HalfEdge` in the inside.

`Face(HalfEdge, HashSet<HalfEdge>)` - The constructor. It will initialize a `Face` bounded on the outside by the first `HalfEdge` and holes defined by the `HashSet` of `HalfEdge` objects.

`equals(Object) : boolean` - Compares whether or not two `Faces` are equivalent by comparing their boundaries and holes.

`GetInner() : HashSet<HalfEdge>` - Returns the set of `HalfEdges` that form holes in the `Face`.

`GetOuter() : HalfEdge` - Returns a `HalfEdge` that bounds the `Face`.

`hashCode() : int` - Returns a hash based on the hashes of the `HalfEdge` components of its outer boundary using a basic hash of prime numbers (31 and 17).

`IsConvex() : boolean` - Returns whether or not a `Face` is convex.

`SetConvex(boolean)` - Sets the convex status of a `Face`.

`SetInner(HashSet<HalfEdge>)` - Sets the set of `HalfEdge` objects that form holes in the `Face`.

`SetOuter(HalfEdge)` - Sets the `HalfEdge` that bounds the `Face`.

`toString() : String` - Returns a `String` representation of the `Face` as a list of its outer and inner boundaries.

5.5 HalfEdge

`HalfEdge()` - The default constructor. It will initialize a `HalfEdge` with no endpoint or `Face`.

`HalfEdge(HalfEdge)` - The copy constructor. It will initialize a `HalfEdge` with the same endpoints and bounded `Face` as the parameter.

`HalfEdge(Vertex, Face, HalfEdge, HalfEdge, HalfEdge)` - The constructor. It will initialize a `HalfEdge` with an origin `Vertex`, a bounded `Face`, a next `HalfEdge`, a previous `HalfEdge`, and a twin `HalfEdge`.

`equals(Object)` : `boolean` - Compares whether or not two `HalfEdge` objects are equivalent by comparing their origins and their twins' origins.

`GetDirection()` : `int` - Returns the direction of the `HalfEdge` chain. The results may be: `-1` for counter-clockwise, `0` for a single `HalfEdge`, or `1` for clockwise.

`GetFace()` : `Face` - Returns the `Face` bounded by the `HalfEdge` on the left.

`GetHelper()` : `Vertex` - Returns the helper `Vertex` for triangulating a polygon.

`GetNext()` : `HalfEdge` - Returns the next `HalfEdge` in the chain that bounds the same `Face`.

`GetOrigin()` : `Vertex` - Returns the origin `Vertex` of the `HalfEdge`.

`GetPrev()` : `HalfEdge` - Returns the previous `HalfEdge` in the chain that bounds the same `Face`.

`GetTwin()` : `HalfEdge` - Returns the twin `HalfEdge` whose origin is the endpoint and whose endpoint is the origin of the current `HalfEdge`.

`hashCode()` : `int` - Returns a hash based on the hashes of the `Vertex` components of its endpoints using a basic hash of prime numbers (31 and 17).

`SetFace(Face)` - Sets the incident `Face` on its left.

`SetHelper(Vertex)` - Sets the helper `Vertex`.

`SetNext(HalfEdge)` - Sets the next `HalfEdge` in the chain.

`SetOrigin(Vertex)` - Sets the origin `Vertex` of the `HalfEdge`.

`SetPrev(HalfEdge)` - Sets the previous `HalfEdge` in the chain.

`SetTwin(HalfEdge)` - Sets the twin `HalfEdge`.

`toString()` : `String` - Returns a `String` representation of the `HalfEdge` as a tuple of the endpoints.

5.6 PersistentRedBlackTreeSet

`PersistentRedBlackTreeSet<D>()` - The default constructor. It will initialize an `PersistentRedBlackTreeSet` and use the default `Comparator` that compares via `hashCode`.

`PersistentRedBlackTreeSet<D>(Comparator<? super D>)` - The constructor. It will initialize a `PersistentRedBlackTreeSet<D>` and use the provided `Comparator<? super D>`.

`contains(Object)` : `boolean` - Returns whether or not the tree set contains an object.

`delete(D)` - Deletes an object from the tree set.

`equals(Object)` : `boolean` - Returns whether or not two trees are equivalent. `get(D)` : `D` - Returns an element if it is in the tree set.

`getElements()` : `List<D>` - Returns a `List` of all of the elements in the tree set sorted in-order traversal.

`getRandomLeaf()` : `D` - Returns a random leaf element in the list in $O(\log n)$ time. Some elements are not stored in leaves, so they will not be sampled.

`GetRootNode()` : `RedBlackNode<D>` - Returns the root node of the tree set.

`hashCode()` : `int` - Returns the hash of the tree.

`insert(D)` : `PersistentRedBlackTreeSet<D>` - Returns the tree with the new data element. The original list remains the same. This is a persistent insertion.

`insertSubTree(PersistentRedBlackTreeSet<D>) : PersistentRedBlackTreeSet<D>`
- Returns the tree with the subtree inserted persistently. All subelements of the subtree remain in the same order. use with caution.

`iterator() : Iterator<D>` - Returns an iterator of the tree for in-order traversal over all of the elements in the tree.

`size() : int` - Returns the number of elements inserted in the tree.

5.6.1 RedBlackTreeNode

`RedBlackNode<D>()` - The default constructor. It will initialize a `RedBlackNode<D>` with no element or children. Its color will be null.

`RedBlackNode<D>(D)` - The constructor. It will initialize a red `RedBlackNode<D>` with a specific element. Its children will be black null nodes.

`RedBlackNode<D>(RedBlackNode<D>)` - The child constructor. It will initialize a black `RedBlackNode<D>` with no element.

`clone() : RedBlackNode<D>` - Returns a shallow clone of the parameter.

`cloneSibling(LinkedList<RedBlackNode<D>>)` - Clones the sibling where the parameter is the list of nodes above in the tree.

`cloneUncle(LinkedList<RedBlackNode<D>>)` - Clones the uncle where the parameter is the list of nodes above in the tree.

`deepClone() : RedBlackNode<D>` - Returns a deep clone of the parameter.

`getElement() : D` - Returns the element in the node.

`getLeft() : RedBlackNode<D>` - Returns the left child of the node.

`getRight() : RedBlackNode<D>` - Returns the right child of the node.

`sibling(LinkedList<RedBlackNode<D>>)` : `RedBlackNode<D>` - Returns the sibling of the node where the parameter is the list of nodes above in the tree.

5.7 Point

`Point(float, float)` - The constructor. It will initialize a `Point` at the given x-y coordinates.

`Cross(Point, Point, Point) : float` - Returns the cross product of the lines extending from the first `Point` object to the other two.

`Dot(Point, Point) : float` - Returns the dot product of the two `Point` objects.

`Dot(Point, Point, Point) : float` - Returns the dot product of the lines extending from the first `Point` object to the other two.

`compareTo(Point) : int` - Returns the relative location of the parameter `Point` object. The possible return values are: -1 is the parameter is to the right or above, 0 if the parameter is at the same location, 1 if the parameter is to the left or below.

`equals(Object) : boolean` - Returns whether or not the two `Point` objects are equivalent.

`getX() : float` - Returns the x-coordinate.

`getY() : float` - Returns the y-coordinate.

`hashCode() : int` - Returns the hash of the `Point` based on a counter system.

`IsBetween(Point, Point) : boolean` - Returns whether or not this `Point` is collinear with the parameters.

`IsLeft(HalfEdge) : boolean` - Returns whether or not this `Point` is to the left of a `HalfEdge`.

`SetX(float)` - Sets the x-coordinate of the `Point`.

`SetY(float)` - Sets the y-coordinate of the Point.

`toString()` - Returns a String representation of the Point as a tuple of the x-y coordinates.

5.8 PointLocation

`PointLocation(DCEL)` - The constructor. It will initialize a PointLocation structure for the associated DCEL.

`GetSlabs()` : List<Slab> - Returns the Slab objects of the PointLocation structure generated.

`Query(Point)` : HalfEdge - Returns a HalfEdge that bounds the Face that contains the Point. Returns null if the Point is not within any internal Face of the DCEL.

5.9 Slab

`Slab(Point, Point)` - The constructor. It will initialize a Slab that has a left bound and right bound.

`AddEdge(HalfEdge)` - Adds a HalfEdge to the Slab.

`GetEdgeBelow(Point)` : HalfEdge - Returns the HalfEdge directly below the Point.

`GetLeftBound()` : Point - Returns the left bound of the Slab.

`GetRightBound()` : Point - Returns the right bound of the Slab.

`IsInSlab(Point)` : boolean - Returns whether or not a Point is within the bounds of the Slab.

5.10 SweepLine

`SweepLine()` - The default constructor. It will initialize a SweepLine with a no Segments.

`AddEvent(Event)` : Segment - Adds an Event to the SweepLine and returns its corresponding Segment.

`FindEvent(Event)` : Segment - Returns the Segment that corresponds to the Event.

`Intersect(Segment, Segment)` : boolean - Returns whether or not the two Segment objects intersect.

`RemoveSegment(Segment)` - Removes a Segment from the SweepLine.

5.10.1 Segment

`compareTo(Segment)` : int - Returns the relative position of the parameter. The possible return values are: -1 if the parameter is below, 0 if the parameter is the same, 1 if the parameter is above.

`equals(Object)` : boolean - Returns whether or not the two Segment objects are equivalent.

`GetAbove()` : Segment - Returns the Segment directly above it.

`GetBelow()` : Segment - Returns the Segment directly below it.

`GetEdge()` : HalfEdge - Returns the HalfEdge corresponding to this Segment.

`GetLeftVertex()` : Vertex - Returns the left endpoint.

`GetRightVertex()` : Vertex - Returns the left endpoint.

`hashCode()` : int - Returns the hash of the corresponding HalfEdge.

`SetAbove(Segment)` - Sets the Segment directly above it.

`SetBelow(Segment)` - Sets the Segment directly below it.

`SetEdge(HalfEdge)` - Sets the HalfEdge corresponding to this Segment.

`SetLeftVertex(Vertex)` - Sets the left endpoint.
`SetRightVertex(Vertex)` - Sets the left endpoint.

5.11 Triangulation

`IsMonotonePolygon(DCEL) : boolean` - Returns whether or not a DCEL is a monotone polygon.
`MakeMonotone(DCEL)` - Adds HalfEdges to the simple DCEL to make a monotone polygon.
`TriangulateMonotonePolygon(DCEL)` - Adds HalfEdges to the monotone DCEL to make a triangulated monotone polygon.

5.12 Vertex

`Vertex()` - The default constructor. It will initialize a `Vertex` with no point, incident edge, UNKNOWN type, and non-convex.

`Vertex(Point)` - A constructor. It will initialize a `Vertex` with a point, but no incident edge, UNKNOWN type, and non-convex.

`Vertex(Point, HalfEdge)` - A constructor. It will initialize a `Vertex` with a point and incident edge, but with UNKNOWN type and non-convex.

`compareTo(Vertex) : int` - Returns the `compareTo` result of the two Points.
`equals(Object) : boolean` - Returns whether or not two `Vertex` objects are equivalent.
`GetEdge() : HalfEdge` - Returns one of the `HalfEdge` incident on this `Vertex`.
`GetPoint() : Point` - Returns the `Point` object of this `Vertex`.
`GetType() : Type` - Returns the `Type` of this `Vertex` for triangulation.
`hashCode() : int` - Returns the hash of the `Point`.
`IsConvex() : boolean` - Returns whether or not this `Vertex` is convex.
`SetConvex(boolean)` - Sets the convex status of this `Vertex`.
`SetEdge(HalfEdge)` - Sets the incident edge of this `Vertex`.
`SetPoint(Point)` - Sets the location of this `Vertex`.
`SetType(Type)` - Sets the `Type` of this `Vertex`.
`toString()` - Returns the `String` representation of this `Vertex` as the tuple of the x-y coordinates.

5.12.1 Type

START - A start `Vertex`.
END - An end `Vertex`.
MERGE - A merge `Vertex`.
SPLIT - A split `Vertex`.
REGULARL - A regular `Vertex` in the left chain.
REGULARR - A regular `Vertex` in the right chain.
UNKNOWN - An uncategorized `Vertex`.

References

[1] Sarnak, Neil; Tarjan, Robert E. *Point location problem with persistent search trees*

- [2] Larkworthy, Tom. *PersistentRedBlackTreeSet* <https://edinburghhacklab.com/2011/07/a-java-implementation-of-persistent-red-black-trees-open-sourced/>