Computational Geometry

INTERSECTION POINTS BETWEEN TWO SIMPLE POLYGONS

PIYUSH RATHI (16CS10004)
SHUBHAM PATIL (16EC10039)

Indian Institute of Technology, Kharagpur
Department of Computer Science and Engineering

Contents

1. Abstract and Problem Description

The paper discusses an algorithm for finding the intersection points of two simple polygons. The search for the intersection points between two simple polygons is one of the fundamental problems of computational geometry. By definition, a simple polygon is any polygon that is not self-intersected (it may be concave). The problem of intersection of polygons edges has been considered by different authors as well as its generalisation: the line-segment intersection problem.

A brute-force approach (trying to intersect each edge of the first polygon with each edge of the second) works with $\mathcal{O}(n \times m)$ time complexity in best as well as average case, where n and m are the respective numbers of vertices of the input polygons.

The algorithm we use is a sweep-line approach which takes use of two one-way dynamic lists of currently pierced line segments. The theoretical worst-case time complexity of this algorithm is still $\mathcal{O}(k^2)$, where $k = (n+m)$, but in practice its run-time is much better. We also generate different test cases that is different pairs of polygons and plot them and their intersection points determined by out algorithm. The implementation for the code can be found here:

2. Review of Prior Work

We use the algorithm put forward by Borut Zalik in June, 1999 under the title "Two efficient algorithms for determining intersection points between simple polygons"[6]. The paper puts forwards two efficient, one of which uses two one-way dynamic lists as the underlying data structure whereas the second one uses binary-search-tree-based data structure. We implement this algorithm using the dynamic lists.

The use of a sweep-line is one of the fundamental techniques in computational geometry[4] and computer graphics[3]. The idea is to divide the plane into vertical sections by the event points where the event points are the vertices of the polygon. While searching for the intersections between polygon edges, only those edges currently being pierced by the sweep-line are

of interest. These edges are good candidates for intersections and they are stored in a data structure usually named a sweep-line status (SLS).[2]

3. Work Plan

First, we define an ordering on all the event points, and sort them according to that ordering. The way ordering is defined as:

1. Based on the x-coordinate of the point.
2. If x-coordinates of two points are same, then tie is broken by which polygon they belong to(we arbitrarily keep the points of first polygon first).
3. If tie still needs to be broken, we break it by the y-coordinate.

When the x-coordinates are same, we break the tie by which polygon they belong to before their y-coordinates because otherwise we would be counting two lines touching each other as intersection which we do not want.

Then, we start processing the event points according to the order set. Let's say we encounter a point v_k with previous neighbor as v_i, and next neighbor as v_j, and the edges connecting them as e_{ki} and e_{kj}, respectively. The rules we encounter are as follows:

1. Rule **A**: $v_i > v_k$ and $v_j > v_k$; insert both edges e_{ki} and e_{kj} into SLS.
2. Rule **B**: $v_i < v_k$ and $v_j < v_k$; remove both edges e_{ki} and e_{kj} from SLS.
3. Rule **C**: $v_i > v_k$ and $v_j < v_k$; insert edge e_{ki} into SLS and remove e_{kj} from SLS.
4. Rule **D**: $v_i < v_k$ and $v_j > v_k$; insert edge e_{kj} into SLS and remove e_{ki} from SLS.
5. Rule **V**: This is a special case when one of previous or next neighbor of the same polygon has the same y-coordinate as the current vertex. Here, we insert e_{kj} into SLS, and based on whether $v_i < v_k$ or $v_i > v_k$, we remove or insert the corresponding edge from SLS, respectively.

Note that after inserting an edge into SLS, we immediately check next whether it intersects any of the pierced edges from the other polygon stored in another SLS.

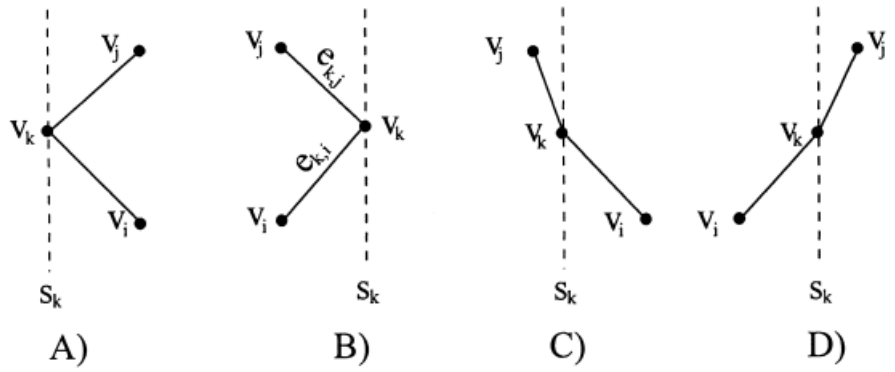

Figure 3.1: Possible positions(A-D) of polygon vertices regarding sweep-line.

4. Formulation and Data Structures

```
struct Point {
    double x, y;
};

struct Polygon {
    vector <Point> points;
    Polygon(int size) {
        points.resize(size);
    }
};
```

Listing 4.1: Data Structures used to store vertices and polygons.

All the vertices are stored in a structure, and the polygon in turn is made from a structure that contains a vector of point structure.

Other data structured used are:

1. A vector of pair point number, polygon number to uniquely identify a vertex.
2. A vector of Points to store intersection points.

3. Two doubly connected linked lists where each of them store the edges of their corresponding polygon that are currently pierced by the sweep line.

After we find out which rule to be used to process the current event point, we use the cases mentioned above to determine which rule to use among 'A', 'B', 'C', 'D', and 'V'. After determining that, we either remove or add some edges to the list corresponding to that polygon.

To remove, we simply remove that edge from the list, and to add we not just add, but after adding we also check which edges of the other polygon intersect with this newly added edge.

To briefly discuss how we check whether or not two line segments intersect, we first find out the intersection point of the two lines, where the lines are just extension of the two line segments to infinity(if there is no intersection point between the two lines, then the line segments do not intersect as well) and then we find out whether the intersection point lies inside both of the line segments or not.[1]

5. Theoretical Results

Let n denote the total number of edges of polygon P and m ($n \geq m$) the total number of edges of polygon Q. For the previously introduced rules, the following estimation of time complexity are obtained:

- Rule A: two edges of a polygon (P or Q) are inserted into the corresponding set and then checked for intersection with edges located in the second set. Both sets are implemented as one way connected lists, and each new entry is appended at the top of the list. This is done in a constant time $\mathcal{O}(1)$. Let us suppose the second set contains r entries. Therefore, the search for the intersections requires $\mathcal{O}(r)$ time.
- Rule B: Removing two edges from a given set with r edges are stored takes $2 \times \mathcal{O}(r)$ time.
- Rule C and D: in this situation, one edge is checked for intersection, and the other one is removed from the corresponding set of edges. Using the estimations mentioned before, this task is also finished in $\mathcal{O}(r)$.
- Rules for solving special cases can be considered using the analogy with rules **A-D**.

Suppose that all n edges of the first polygon have been already inserted in the first set. Then we insert step-by-step all m edges of the second polygon and perform a check for intersections with the n edges stored in the first set. This gives:

$$T = m \times n = \mathcal{O}(n \times m) = \mathcal{O}(n^2) \tag{5.1}$$

However, in general, the length of the second set is smaller than n ($r < n$) giving us

$$T = r \times n = \mathcal{O}(r \times n) \tag{5.2}$$

If $r \ll n$, T tends even to the linear time complexity $\mathcal{O}(n)$ which is achieved if polygons do not intersect at all.

6. Experimental Analysis

In this section, we study the performance of the proposed algorithm. The configuration of our experiment is illustrated first, which includes generation of testing data, evaluation metrics and implementation settings. Finally, we analyze the run-time of the proposed algorithm [1].

6.1 Generation of Random Polygons

To test the performance of proposed algorithm, testing data is needed. Here we deal with the random generation of simple polygons on a given set of points. Two methods are used to generate a random set of points $\mathcal{S}$, which are as follows:

1. x and y co-ordinates are generated uniformly between two given parameters min and max.
2. x and y co-ordinates are generated by normal distribution with given standard deviation σ and mean μ.

Given a set $\mathcal{S} = \{s_1, ..., s_n\}$ of n points, we would like to generate a simple polygon $\mathcal{P}$ at random with a uniform distribution. In this context, a uniformly random polygon on $\mathcal{S}$ is a polygon which is generated with probability $\frac{1}{k}$ if there exist k simple polygons on $\mathcal{S}$ in total.

On a given random set of points $\mathcal{S}$, to generate a simple polygon we use *Space Partitioning* method [5]. Space partitioning recursively partitions $\mathcal{S}$ into subsets which have disjoint convex hulls. Let $\mathcal{S}^{'}$ be a such a subset of $\mathcal{S}$. (Thus, $\mathcal{CH}(\mathcal{S}^{'})$ does not contain any point of $\mathcal{S}\backslash\mathcal{S}^{'}$.) When generating a polygon $\mathcal{P}$ we will guarantee that the intersection of $\mathcal{P}$ with $\mathcal{CH}(\mathcal{S}^{'})$ consists of one single chain. The first point of this chain is denoted by $s_f^{'}$, and its last point by $s_l^{'}$. Note that both $s_f^{'}$ and $s_l^{'}$ are located on the boundary of $\mathcal{CH}(\mathcal{S}^{'})$. The pseudo code for space partitioning method is presented in Algorithm 1. During the initial phase of the algorithm, we choose $s_f, s_l \in \mathcal{S}$ at random. Space Partitioning method can compute the simple polygon in at most $\mathcal{O}(n^2)$ time.

6.2 Implementation Details

We implemented our algorithms together with a test bed in the programming language C++. The hardware platform running experiments is Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz-

[1] Codes are available on https://github.com/fsociety88/Computational-Geometry-Project

Algorithm 1 Space-Partitioning-Algorithm

Require: $\mathcal{S}, s_f, s_l$
1: **if** s_f and s_l are the only points of $\mathcal{S}$ **then**
2: **return** Line segment $s_f s_l$.
3: **end if**
4: Choose a point $s \in \mathcal{S}$ at random.
5: Select a random line l through s such that l intersects $s_f s_l$.
6: $\mathcal{S}_L$, $\mathcal{S}_R$ are left and right sets resp. partitioned by line l.
7: $\mathcal{P}_L$ = Space-Partitioning-Algorithm($\mathcal{S}_L$, s_f, s)
8: $\mathcal{P}_R$ = Space-Partitioning-Algorithm($\mathcal{S}_R$, s, s_l)
9: **return** $\mathcal{P}_L \cup \mathcal{P}_R$

2.90GHz, 8GB RAM, 1TB hard drive, 64-bit Windows 10 operating system.

6.3 Results

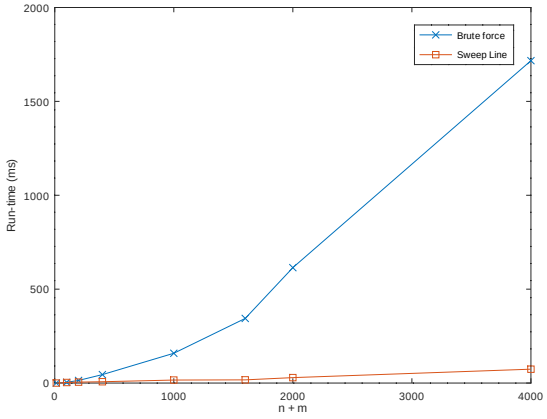

Figure 6.1: Diagram of spent CPU time with respect to number of polygon edges.

Engineers are often not interested in the pure theoretical estimation of an algorithm time complexity, because it does not necessarily give the best estimate of computation time. The most valuable guide comes from measurements of the spent CPU time for a given data set. We have compared our algorithm with a brute force algorithm. The brute force algorithm is the most primitive implementation. It does not introduce any test to minimise the number of edges being checked for intersection. We give run-time of suggested algorithm in Table 6.1 and 6.2.

In this algorithm, the determination of whether any pair of polygon edges intersect is the most critical time factor especially if no intersection points exist. We compared the results for two different data sets which were generated by two different methods.

1. For the first set of test data, we generated points by randomly selecting x and y co-ordinates between the range *(MIN, MAX)*. Without loss of generality, we assumed that

Table 6.1: Measurement of spent CPU time (in ms). Generated testing data is uniformly distributed between MIN and MAX.

Δ *MAX - MIN*	n	m	Avg. Run-time (in ms) Brute Force	Sweep Line
10	5	5	0	0
	50	50	2.99	0.996
	100	100	7.979	0.997
	200	200	21.948	2.994
	500	500	95.768	8.978
	800	800	271.343	18.952
	1000	1000	563.64	27.966
	2000	2000	1661.02	62.849
50	5	5	0	0
	50	50	4.033	2.985
	100	100	13.967	4.984
	200	200	44.936	6.986
	500	500	158.616	15.959
	800	800	344.169	16.945
	1000	1000	614.517	28.929
	2000	2000	1716.86	73.853
100	5	5	0.992	0
	50	50	2.985	0
	100	100	9.967	1.995
	200	200	25.961	2.991
	500	500	99.784	16.959
	800	800	274.361	14.952
	1000	1000	547.702	33.918
	2000	2000	1634.05	65.84
500	5	5	0.997	0
	50	50	2.992	0
	100	100	9.974	0.966
	200	200	33.916	2.993
	500	500	126.692	9.975
	800	800	311.246	16.96
	1000	1000	579.6	27.898
	2000	2000	1662.04	77.778

$MIN = 0$. We compared the run-time of proposed algorithm with brute force for four different values of $\Delta = MAX - MIN = \{10, 50, 100, 500\}$. Polygons were constructed using space partition method 5.1. For each value of Δ, we generated 8 different test data, $(n, m) = \{(5, 5), (50, 50), (100, 100), (200, 200), (500, 500), (800, 800), (1000), (1000), (2000, 2000)\}$. Test data were executed on both brute force and the sweep line algorithm. Table 6.1 shows the results.

2. For the second set of test data, we generated x and y co-ordinates of vertices of polygons normally with μ and σ. Without loss of generality, we assumed that $\mu = 0$. We

Table 6.2: Measurement of spent CPU time (in ms). Generated testing data is normally distributed with $\mu = 0$ and σ.

σ ($\mu = 0$)	n	m	Avg. Run-time (in ms) Brute Force	Sweep Line
10	5	5	0	0
	50	50	3.989	0.999
	100	100	12.965	2.992
	200	200	39.891	4.987
	500	500	122.67	18.95
	800	800	298.232	33.89
	1000	1000	604.414	91.756
	2000	2000	1700.48	94.744
20	5	5	0	0
	50	50	1.961	1.018
	100	100	6.977	1.995
	200	200	20.917	3.989
	500	500	94.713	14.96
	800	800	271.242	18.979
	1000	1000	539.525	26.959
	2000	2000	1631.61	76.797
50	5	5	1.025	0
	50	50	3.028	0.997
	100	100	9.008	1.994
	200	200	23.964	2.958
	500	500	100.73	9.973
	800	800	277.257	20.004
	1000	1000	546.538	34.906
	2000	2000	1635.63	71.772
100	5	5	0	0
	50	50	2.991	0.998
	100	100	9.973	1.994
	200	200	26.928	2.993
	500	500	108.709	9.972
	800	800	286.268	18.951
	1000	1000	560.501	28.954
	2000	2000	1667.54	69.847

compared run-time of proposed algorithm with brute force for four different values of $\sigma = \{10, 20, 50, 100\}$. For each value of σ, we generated 8 different polygons in similar way as the first set of test data. Table 6.2 shows the results.

The number of vertices in both polygons were increased gradually. Figure 6.1 shows the comparison between run-times of brute-force implementation and the sweep line algorithm. Figure 6.2 shows the pictorial representation of the result returned by the algorithm for a test case.

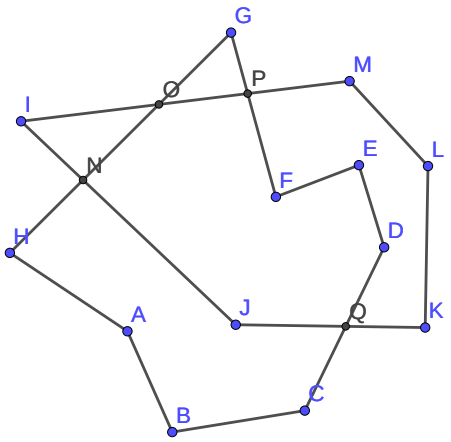

Figure 6.2: Polygon intersection points. Two polygons are randomly generated ($n = 8, m = 5$). The points of intersection returned by sweep line algorithm are N, Q, P, O.

7. Discussion

In the suggested algorithm, the determination of whether any pair of polygon edges intersect is the most critical time factor especially if no intersection points exist. From the results, it can be seen how fast the suggested algorithm is compared to brute force implementation (see Table 6.1, 6.2 and diagram in Fig. 6.1). It can be concluded that sweep line algorithm is efficient, although its theoretical time-complexity evaluation suggests that it is not much better that the brute-force algorithm.

8. Conclusion

The classical problem of computational geometry which considers the search for intersection points between the edges of two polygons appears in many applications of computer graphics (hidden-line, hidden-surface elimination), robotics (detection of collision of robot arm with the neighbouring objects), and in GIS. Usually, the brute-force approach is implemented, which is inefficient.

The algorithm for determination of intersection points between polygon edges, using a sweep-line, is suggested in the report. The algorithm has $\mathcal{O}(k^2)$, $k = n + m$, time complexity in the worst case. The theoretical time complexity is approximate and suggest that the actual run-time should be related to the brute-force implementation (both have the same theoretical time complexity $\mathcal{O}(k^2)$). As shown in practice (see Table 6.1, 6.2 and diagram in Fig. 6.1), however, the CPU time is much faster than the brute-force implementation.

Bibliography

[1] How to check if two given line segments intersect? **Link**.

[2] Herbert Edelsbrunner Bernard Chazelle. An optimal algorithm for intersecting line segments in the plane. 1992.

[3] van Dam A.-Feiner S.K. Hughes J.F. Foley, J.D. Computer graphics: Principles and practice, 2nd ed. addison-wesley, reading, massachusetts, 1174 pp.m. 1990.

[4] Shih T.-Y. Huang, C.-W. On the complexity of pointin-polygon algorithms. computers geosciences 23 (1), 109-118. 1997.

[5] Martin H Thomas A. Heuristics for the generation of random polygons. 1996.

[6] Borut Žalik. Two efficient algorithms for determining intersection points between simple polygons. 2000.