RE2L: An Efficient Output-sensitive Algorithm for Computing Boolean Operations on Circular-arc Polygons and Its Applications

Zhi-Jie Wang[a,b], Xiao Lin[c], Mei-E Fang[d], Bin Yao[a], Yong Peng[d], Haibing Guan[a], Minyi Guo[a]

[a]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China*
[b]*Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong*
[c]*Department of Science and Engineering, University of Shanghai for Science and Technology, Shanghai, China*
[d]*School of Computer Science, Hangzhou Dianzi University, Hangzhou, China*

Abstract

The boundaries of *conic polygons* consist of conic segments or second degree curves. The conic polygon has two degenerate or special cases: the linear polygon and the circular-arc polygon. The natural problem — boolean operation on linear polygons, has been *extensively* studied. Surprisingly, (almost) no article *focuses on* the problem of *boolean operation on circular-arc polygons*, which actually also has many applications. This implies that if there is a targeted solution for boolean operation on circular-arc polygons, it should be favourable for potential users. In this article, we close the gap by devising a concise data structure and then developing a targeted algorithm called RE2L. Our method is simple, easy-to-implement but without loss of efficiency. Given two circular-arc polygons with m and n edges respectively, our method runs in $O(m+n+(l+k)\log l)$ time, using $O(m+n+k)$ space, where k is the number of intersections, and l is the number of related edges (defined in Section 5). Our algorithm has the power to approximate to linear complexity when k and l are small. The superiority of the proposed algorithm is also validated through empirical study. Particularly, our method is of independent interest, we show it can be easily extended to compute boolean operation of other types of polygons.

Keywords: Boolean operations, circular-arc polygons, related edges, sequence lists, appendix points

1. Introduction

Boolean operation on polygons is one of the oldest and best-known problems in computer graphics, and it has attracted much attentions, due to its simple formulation and broad applications in various disciplines such as computational geometry, CAD, GIS, visual computing, motion planning [5, 8, 9, 12, 18, 27–29]. When the polygons to be operated are *conic polygons* (whose boundaries consist of conic segments or second degree curves), researchers have made some efforts, see, e.g., [3, 4, 10]. The conic polygon has several special or degenerate cases: (i) the linear polygon (known as traditional polygon), whose boundaries consist of only linear curves, i.e., straight line segments; and (ii) the circular-arc polygon, whose boundaries consist of circular-arcs and/or straight line segments. The natural problem — boolean operation on traditional polygons — has been extensively investigated, see e.g., [1, 11, 13–18, 20–22, 25]. However, in existing literature (almost) no article *focuses on* another natural problem — boolean operation on circular-arc polygons. In fact, boolean operation on circular-arc polygons also has many applications. For instance, deploying sensors to ensure wireless coverage is an important problem [19, 26]. The sensing range of a single sensor is a circle. With polygonal obstacles, its sensing range is cut off, shaping a circular-arc polygon. When we verify the wireless coverage range of sensors, boolean operation on circular-arc polygons is needed. As another example, assume there are a group of free-rotating cameras used to monitor a supermarket. The visual range of a single camera can be regarded as a circle, as it is to be freely rotated. Various obstacles such as goods shelves usually impede the visions of cameras, here boolean operation can be used to check the blind angles. Last but not least, consider a group of free-moving robots used to guide the visitors in a museum. Since the energy of a single robot is limited, its movable region is restricted to a circle. With the impact of various obstacles such as exhibits, the original movable region is to be cut off by these obstacles. When we verify if every place in the museum can be served by at least a robot, boolean operation on circular-arc polygons is also needed.

Although the solution used to handle conic (or more general) polygons can also work for its special cases, the special cases, however, usually have their unique properties; directly executing the algorithm used to handle conic (or more general) polygons is usually not efficient enough. It is just like when the applications only involve traditional polygons, we usually incline to use the solutions targeted for traditional polygons rather than the ones for conic (or more general) polygons.

Email addresses: zjwang888@sjtu.edu.cn (Zhi-Jie Wang), lin6008@126.com (Xiao Lin), fme@hdu.edu.cn (Mei-E Fang), yaobin@cs.sjtu.edu.cn (Bin Yao), standy.peng@gmail.com (Yong Peng), hbguan@cs.sjtu.edu.cn (Haibing Guan), guo-my@cs.sjtu.edu.cn (Minyi Guo)

With the similar argument, when the applications only involve circular-arc polygons, a targeted solution for circular-arc polygons should be favourable for potential users.

Motivated by these, this paper makes the effort to the problem of *boolean operation on circular-arc polygons*. In particular, we are interested in developing algorithms with the following features: (i) easy-to-implement for deployment in practice, and (ii) having nice theoretical guarantees. To this end, first of all a concise and easy-to-operate data structure is naturally developed (Section 4.1); based on this concise structure, we then propose an algorithm dubbed as RE2L that consists of three main steps.

The first step is the kernel (or core) of RE2L, yielding two *special* sequence lists. Specifically, the kernel integrates three simple yet efficient strategies: (i) it introduces the concept of **r**elated **e**dges, which is used to avoid irrelevant computation as much as possible; (ii) it employs two *special* sequence **l**ists, each one is a compound structure with three domains; they are used to let the *decomposed arcs*, intersections and *processed related edges* be well organized, and thus immensely simplify the subsequent computation; and (iii) it assigns two **l**abels to each processed related edge before the edge is placed into a balanced tree; this contributes to avoiding the "false" intersections being reported, and speeding up the process of inserting the reported intersections into their corresponding edges (Section 5). The second step produces two *new* linked lists in which the intersections, appendix points, and original vertices have been arranged, and the decomposed arcs have been merged. To obtain these two new linked lists, two important but easy-to-ignore issues, "inserting new appendix points" and "merging the decomposed arcs", are addressed (Section 6). The third step is to obtain the resultant (or output) polygon by traversing these two new linked lists. In order to correctly traverse them, the *entry-exit* properties are naturally adopted, and three traversing rules are developed (Section 7).

Viewed from a macro perspective, similar to many methods (see e.g., [1, 11, 14, 15, 24, 25]) in the literature, our solution also partially inherits two well-known proposals: Bentley-Ottmann Plane Sweep algorithm [2] and Weiler-Atherton Clipping algorithm [29], whereas we also advance existing results from various aspects. To summarize, we make the following main contributions.

1. We highlight the circular-arc polygon is one of special cases of the conic polygon, and boolean operation on circular-arc polygons also has many applications.
2. We devise a concise and easy-to-operate data structure, and develop a targeted algorithm for boolean operations on circular-arc polygons.
3. While this paper focuses on boolean operations of circular-arc polygons, we show our techniques can be easily extended to compute boolean operations of other types of polygons (Section 9).
4. We provide the rigorous and detailed theoretical analysis for our algorithm. In brief, given two circular-arc polygons with m and n edges respectively, our algorithm runs in $O(m+n+(l+k)\log l)$ time, using $O(m+n+k)$ space, where k is the number of intersections, and l is the number of related edges (Section 8).
5. We conduct extensive experiments to demonstrate the efficiency and effectiveness of our solution (Section 10).

The novelty of our work is threefold: to the best our knowledge (i) it is the first comprehensive study on boolean operations of circular-arc polygons; (ii) it is the first time to employ the idea "utilizing related edges" for boolean operations of polygons, this technique is simple enough to be practical value; and (iii) it is the first output-sensitive algorithm having the potential to approximate to linear complexity for boolean operations of polygons.

Next, we review previous works most related to ours, and then present our algorithm including rigorous theoretical analysis and extensive empirical study.

2. Related Work

We first clarify several technical terms for ease of presentation. It is well known that there are three typical boolean operations: intersection, union, and difference. Note that *polygon clipping* mentioned in many papers is actually to compute the *difference* of two polygons [25]. Given two polygons, the one to be clipped is called the *subject polygon*, another is usually called the *clip polygon* or *clip window* [14, 24, 25]. Given a polygon, if there is a pair of non-adjacent edges intersecting with each other, this polygon is usually called the *self-intersection* polygon [1, 11, 22]. Throughout this paper, the *traditional polygon* refers to the polygon whose boundaries consist of *only* straight line segments, while the *circular-arc polygon* refers to the polygons whose boundaries consist of circular arcs, or both straight line segments and circular arcs. We are now ready to review the previous works most related to ours.

2.1. Boolean Operations on Traditional Polygons

In existing literature, there are many papers studying boolean operation of traditional polygons. For example, Sutherland-Hodgeman [24] proposed an elegant algorithm dealing with the case when the *clip polygon* is convex. Liang et al. [14] gave elaborate analysis on the case when the *clip polygon* is rectangular. Andreev [1] presented an algorithm dealing with the case when the *subject polygon* is with holes and self-intersections. Vatti [25] and Greiner-Hormann [11] proposed general algorithms that can handle concave polygons with holes and self-intersections for both the clip and the subject polygons. Later, Liu et al. [15] further optimized Greiner-Hormann's algorithm. Rivero-Feito [22] achieved boolean operation of polygons based on the concept of *simplicial chains*. Peng et al. [20] also adopted simplicial chains, and improved Rivero-Feito's algorithm. Recently, Martinez et al. [18] proposed to subdivide the edges at the intersection. These works lay a solid foundation for the future research. Compared to these works, this paper focuses on boolean operation of circular-arc polygons, and thus is different from theirs.

2.2. *Boolean Operations on Conic/General Polygons*

Researchers have also made some efforts on boolean operations of conic polygons. For example, Berberich et al. [3] proposed to decompose non-x-monotone curves and compute the arrangement of segments using the plane sweep method, and then compute the *overlap* of two polygons using the results of arrangement, in order to achieve boolean operations. Gong et al. [10] achieved boolean operation of conic polygons using the topological relation between two conic polygons, this method does not require *x-monotone* conic arc segments. Both algorithms can support boolean operation of circular-arc polygons, as the conic polygon is the general case of the circular-arc polygon. Moreover, the computational geometry algorithms library (CGAL) [7] can also support boolean operation of circular-arc polygons. Inspecting the source codes of CGAL, we realize that its idea is directly invoking the algorithm of boolean operation on *general polygons*, defined as `General Polygon_2` in CGAL[1]. To some extent the general polygon can be considered as the most general case, as its edges can be line segments, circular arcs, conic curves, cubic curves, or even more complicated curves. Although the essence of the algorithm in CGAL is basically similar to that in [3] (using the *plane sweep method* to compute the intersections, and the DCEL structure to represent the polygons), CGAL is a very powerful and useful library collecting many classical ideas. For example, Emiris et al. [6] developed a kernel, for curved objects and related operations, that was targeted for inclusion in CGAL. Note that, the CGAL project itself also yields many nice papers in which boolean operation on polygons with curves is mentioned, see e.g., [4, 30], to mention just a few. These excellent works are the cornerstones of our study, giving us a lot of inspiration.

Compared to these works, our work is different from theirs in the following aspects at least. First, this paper focuses on one of special cases of conic polygon; specially, we give insights into its unique properties, design a concise data structure customized for this special case, and develop a targeted algorithm, in which the central idea 'utilizing related edges' (accompanied with a set of well-designed strategies) is proposed. To our knowledge, it is the first time to employ this technique for boolean operations on polygons. Moreover, we give the rigorous theoretical analysis for our algorithm, which runs in $O(m+n+(l+k)\log l)$ time, and approximates to linear complexity when k and l are small (notice: the best known result for polygon boolean operation runs in $O((m+n+k)\log(m+n))$ time, which is no better than *linearithmic time*[2] even if k is small); its superiority is also verified by extensive experiments.

3. Solution Overview

This section describes our algorithm at a high level. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be the circular-arc polygons to be operated, and E_1 (resp., E_2) be the set of all edges of $\mathcal{P}_1$ (resp., $\mathcal{P}_2$), and $\mathcal{P}_3$ be the resultant polygon (i.e., the output of our algorithm). The overall framework of our solution is illustrated in Figure 1. It contains three main steps: (1) construct two sequence lists; (2) build two new linked lists; and (3) traverse two new linked lists. Note that, the first step is the kernel of our algorithm, as the central idea "utilizing related edges" (accompanied with several other helpful strategies) is included in this step.

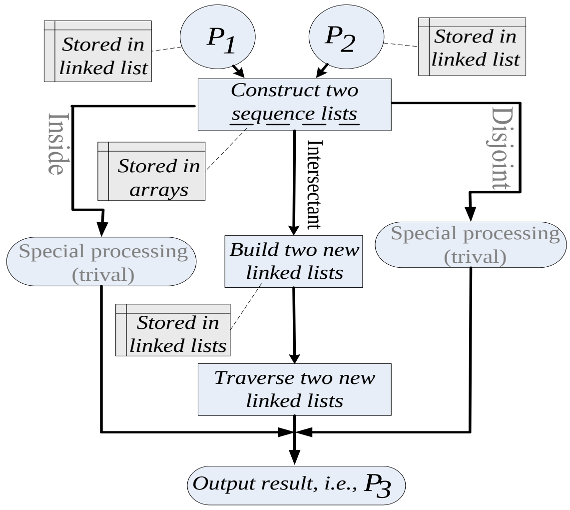


Figure 1: Framework of our solution. The term "Inside" refers to the case: $\mathcal{P}_1 \subseteq \mathcal{P}_2$ or $\mathcal{P}_2 \subset \mathcal{P}_1$. The term "Disjoint" refers to the case: $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$. The term "Intersectant" refers to the case: $\mathcal{P}_1 \cap \mathcal{P}_1 \neq \emptyset$.

Before explaining the construction process of two *sequence lists* (i.e., arrays), we first understand their internal structures. In brief, each item in the two sequence lists is a compound structure consisting of multiple domains; these domains are used to store various information such as edges and intersections (Section 5.2). The essence of constructing two sequence lists is to choose a set E_1' of edges from E_1 and another set E_2' of edges from E_2 (Section 5.1), and compute the intersections of these edges based on a *modified* plane sweep method (Section 5.3). Note that, some arcs (i.e., edges) in E_1' and E_2' may need to be decomposed before computing the intersections. These edges, intersections, and many other information are stored in the two sequence lists orderly, completing Step 1. It is noteworthy that in the construction phase, we can determine the geometry relation between $\mathcal{P}_1$ and $\mathcal{P}_2$, based on some available information. It is trivial to get the resultant polygon when $\mathcal{P}_1$ and $\mathcal{P}_2$ are disjoint (or when one is inside another). In the subsequent discussion, we focus our attention on the case when $\mathcal{P}_1$ and $\mathcal{P}_2$ intersect with each other, for ease of exposition.

We proceed to explain Step 2 "build two *new linked lists*". We need to mention that the input polygon $\mathcal{P}_1$ ($\mathcal{P}_2$) is also stored using the linked list like data structure. Unless stated otherwise, the terms *"input polygons"* and *"original linked lists"* are used interchangeably in the rest of the paper. Note that, although the original linked lists and the new linked lists use the same data structure, the information stored in them is different. To understand Step 2, consider that we have two sequence lists (obtained in the previous step) and two original linked lists (i.e., input polygons). The task is to efficiently merge part of information in the original linked lists and the information in the two se-

[1] More information please refer to the site: http://www.cgal.org

[2] Simply speaking, linearithmic time in Big O notation refers to $O(N \log N)$, provided that the input is $O(N)$ size.

quence lists, and store the 'merged information' using two new linked lists. Particularly, for the two new linked lists we are asked to maintain some properties: for example, the intersections and original vertices should be well arranged, the decomposed arcs should be merged (Section 6). Note that although Step 2 partially inherits the well-known Weiler-Atherton Clipping method, the problem considered here is much more challenging, and thus needs more treatments. On the other hand, as we generate two sequence lists in the previous step, the detailed steps of building the two linked lists are basically different from Weiler-Atherton's and also other existing methods (see e.g., [1, 11, 14, 15, 24, 25]).

Step 3 is essentially adapting existing traversing methods that were used to compute boolean operations of traditional polygons. Briefly speaking, it first assigns *entry-exit* properties to intersections (Section 7.1), and then traverses the two new linked lists based on various traversing rules (Section 7.2). Our paper discusses various boolean operations (intersection, union and intersection), and investigates various circular-arc polygons: with or without holes, self-intersection or non self-intersection (Sections 7 and 9). It is non-trivial and also meaningful to provide a comprehensive investigation, we believe it would be helpful for interested readers.

In the successive sections, we describe in detail our solution including data structure, main ideas, algorithms, and rigorous theoretical analysis.

4. Preliminary

4.1. Data Structure

It is well known that the traditional polygon can be represented by a series of vertices. This method however is invalid for polygons containing circular arcs, as two vertices cannot exactly determine a circular arc segment (note: it may be a major or minor arc). Even so, this ambiguity can be easily eliminated by adding an *appendix point*, where the appendix point can be an arbitrary point that is located on the arc but it is not the endpoints of the arc. For clarity, a traditional vertex is denoted by v_i, and an appendix point is denoted by $\widetilde{v_j}$. For example, $\{v_1, \widetilde{v_2}, v_3, v_4, v_5\}$ determine a circular-arc polygon with four edges (including one circular arc segment $\widehat{v_1\widetilde{v_2}v_3}$ and three straight line segments $\overline{v_3v_4}$, $\overline{v_4v_5}$, $\overline{v_5v_1}$). Unless stated otherwise, in the rest of the paper we always use $\overline{\cdot}$ and $\widehat{\cdot}$ to denote the line segment and the arc segment, respectively. In order to efficiently operate circular-arc polygons, we devise a data structure called APDLL (appendix point based doubly linked list). Specifically, each node in the list consists of several domains below.

- Data: (x, y), the coordinates of a point.
- Tag: Boolean type, it indicates whether this point is a traditional vertex or an appendix point.
- Crossing: Boolean type, it indicates whether this point is an intersection.
- EE: Boolean type, it indicates what property (*entry* or *exit*) an intersection has.
- Prev: Node pointer, it points to the previous node.
- Next: Node pointer, it points to the next node.

4.2. Observation

In this subsection, we introduce a simple yet important observation that will be frequently used later. To explain, we need some preliminaries.

Definition 4.1. *(*Non-x-monotone *circular arc) Given any circular arc, it is a non-x-monotone circular arc such that there is at least one vertical line that intersects with the circular arc at two points.*

Definition 4.2. *(*X-monotone *circular arc) A circular arc is an x-monotone circular arc such that there is at most one intersection with any vertical line.*

Lemma 4.1 below formalizes our observation, which can be viewed as a unique property of circular-arc polygons (compared to other types of polygons).

Lemma 4.1. *Let N_{xmc} be an arbitrary non-x-monotone circular arc, and C be its corresponding circle. Assume that l_h is a horizontal line passing through the center of C. We have that l_h can decompose N_{xmc} into at least two and at most three x-monotone arcs.*

Proof. It is immediate by *analytic geometry*. See Figure 2 for some illustrations. ☐

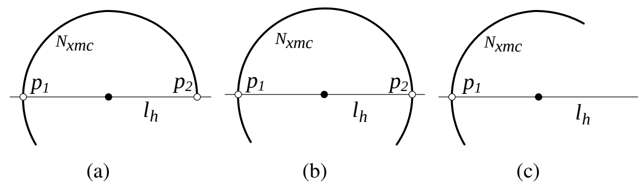


Figure 2: Illustrations of Lemma 4.1.

5. The Kernel of RE2L

In this section, we detail Step 1 of our solution. Specifically, we first expatiate the main ideas integrated in Step 1 (Sections 5.1-5.3), and then present the detailed algorithms (Section 5.4).

5.1. Utilizing Related Edges

One of our strategies is to choose *related edges* (defined later) before doing others. The purpose of choosing *related edges* is to avoid operations that are irrelevant with obtaining the final result as much as possible. To define related edges formally, we need two notions.

Definition 5.1. (Extended boundary lines) *Given a circular-arc polygon, w.l.o.g. (without loss of generality), assume the coordinates of left-bottom corner of its MBR (minimum bounding rectangle) are (x_1,y_1), the one of right-top corner are (x_2,y_2). Then, the following four lines, $X{=}x_1$, $X{=}x_2$, $Y{=}y_1$, $Y{=}y_2$ are respectively the left, right, bottom and top extended boundary lines of this circular-arc polygon.*

Definition 5.2. (Effective axis) *Let I_{mm} be the intersection set of two circular-arc polygons' MBRs. If the horizontal span of I_{mm} is larger or equal to its vertical span; then, the y-axis is the effective axis. Otherwise, the x-axis is the effective axis.*

We now provide the formal definition and inspect more properties of related edges.

Definition 5.3. (Related edges) *Let l_1(l_2) and r_1(r_2) be the left and right extended boundary lines of the circular-arc polygon $\mathcal{P}_1$($\mathcal{P}_2$), respectively. W.l.o.g., assume the effective axis is the x-axis and $l_1 < l_2 < r_1 < r_2$, where $l_1 < l_2$ denotes l_1 is in the left of l_2. Then, the following edges are related edges: (i) edges located between l_2 and r_1; or (2) edges intersected with l_2 or r_1.*

See Figure 3(a) for an example, edges $\overline{ab}$ and $\overline{bc}$ are *related edges* as they intersect with l_2. Similarly, edges $\overline{de}$ and $\overline{ef}$ are also *related edges*. We remark that in Definition 5.3 there are actually other cases, e.g., "$l_1 < l_2 < r_2 < r_1$" or the effective axis is the y-axis; these cases are similar to the listed case, omitted.

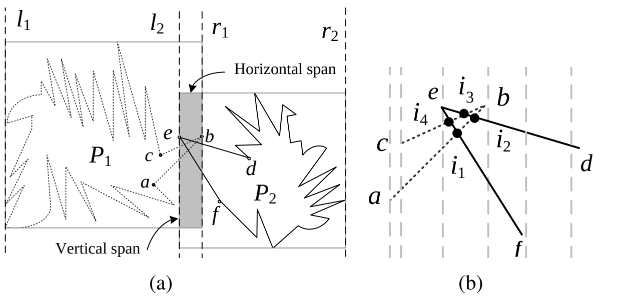


Figure 3: Example of *related edges*. (a) Two big rectangles denote the MBRs; the grey rectangle denotes the intersection set of two MBRs, and the dashed vertical lines denote the extended boundary lines. (b) Partial enlarged drawing.

Definition 5.4. (Processed related edges) *Given a number of related edges, we decompose them if there are non-x-monotone arcs, we call all the edges (after decomposing) the processed related edges.*

By Lemma 4.1 and Definition 5.4, we have the following corollary (which will be used later).

Corollary 5.1. *Given l related edges, if there is no non-x-monotone arc among them, the number of processed related edges is l. Otherwise, the number of processed related edges is larger than l and no more than $3l$.*

Up to now, we have discussed the properties of related edges, and briefly explained how to choose related edges from two circular-arc polygons (remark: more explanations will be given in Algorithm 1 and in the proof of Lemma 5.1). We next show how to use two sequence lists to manage the *processed related edges* and other important components.

5.2. Managing Important Components

The main purpose of the two sequence lists (i.e., arrays) is to let the processed related edges, intersections and decomposed arcs be well organized, which can facilitate the subsequent operations. Specifically, each item in the two sequence lists is a compound structure consisting of three domains: (i) the processed related edge; (ii) the intersections (if exist) on this edge; and (iii) a tri-value switch. For ease of discussion, we denote by S_1 and S_2 the two sequence lists, by $S_i[j]$ the jth item in S_i ($i \in 1,2$), and by $S_i[j].a$, $S_i[j].b$ and $S_i[j].c$ the three domains of $S_i[j]$, respectively.

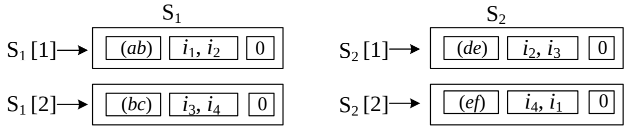


Figure 4: Example of *sequence lists*.

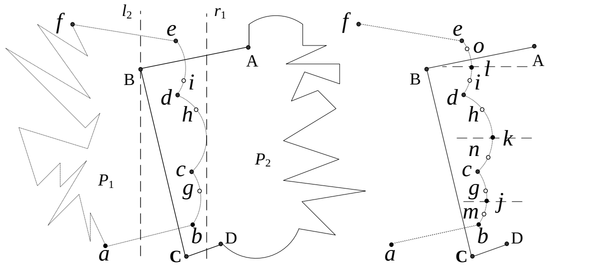
The processed related edges in each sequence list are stored in counter-clockwise direction with regard to the original circular-arc polygon. For example, regarding to circular-arc polygons in Figure 3, we construct two sequence lists as shown in Figure 4. Note that when there are multiple intersections on an edge, we should keep these intersections in order. See $S_1[1].b$ of Figure 4 for an example, the point i_1 is ahead of the point i_2. Regarding to the third domain $S_i[j].c$, it is assigned to either 0, 1, or 2. The assignment rules are as follows. When the edge is not a decomposed arc, we assign "0" to $S_i[j].c$. In this example, for any $1 \leq j \leq |S_i|$ (where $|\cdot|$ denotes the cardinality of S_i), $S_i[j].c$ is set to 0, as there is no decomposed arc. Otherwise, we assign "1" or "2" to $S_i[j].c$. The readers may be curious why we use two different values. The purpose is to differentiate the decomposed arcs which are from different non-x-monotone arcs. This can help us efficiently merge them in the future. (The specific steps on how to merge them will be discussed in Section 6.) Given a series of decomposed arcs, we assign "1" to each decomposed arc that is from the *odd* (1st, 3rd, $\cdots$) non-x-monotone arc, and assign "2" to each decomposed arc that is from the *even* (2nd, 4th, $\cdots$) non-x-monotone arc.

See Figure 5(a) for an example, there are five *related edges* in $\mathcal{P}_1$. Furthermore, Figure 5(b) illustrates eight *processed related edges* (after we decompose them based on Lemma 4.1), implying that $|S_1| = 8$. Based on the assignment rules, the values of the third domains should be "0, 1, 1, 2, 2, 1, 1, 0", respectively.

So far, we have shown how to use two sequence lists to manage the processed related edges and intersections. Note that, in order to obtain the intersections, a standard technique is the *plane sweep method* [2, 23]. In this paper, we do not directly use this algorithm. Instead, we modify it by adding *two labels* to avoid the "false" intersections being reported, and to speed up the process of inserting the reported intersections into their corresponding edges. (Remark: here the *false* intersections refer to the vertices of polygons.) We next give a brief summary of the plane sweep algorithm, and then show how the two labels work.

Plane sweep method. Let $\mathcal{Q}$ be a priority queue, $\mathcal{R}$ be a balanced tree[3], and l_v be a vertical sweep line. The basic idea of the plane sweep method is as follows. First, it sorts the endpoints of all segments according to their x-coordinates, and then puts them into $\mathcal{Q}$. Next, it sweeps the plane (from left to right) us-

[3]It is not mandatory to use a priority queue and a balanced tree, whereas they are usually being recommended, for the sake of efficiency [2]. Moreover, both of them are abstract concepts; the priority queue, for example, can be implemented with a heap or other methods.



(a) (b)

Figure 5: Example of consecutive non-x-monotone circular arcs. (a) Edges $\overline{ab}$, $\widehat{bgc}$, $\widehat{chd}$, $\widehat{die}$, $\overline{ef}$ are *related edges* of $\mathcal{P}_1$. (b) Edges $\overline{ab}$, $\widehat{bmj}$, $\widehat{jgc}$, $\widehat{cnk}$, $\widehat{khd}$, $\widehat{dil}$, $\widehat{loe}$, $\overline{ef}$ are *processed related edges of* $\mathcal{P}_1$; three dashed lines are the auxiliary lines.

ing l_v. At each endpoint during this sweep, if an endpoint is the left endpoint of a segment, it inserts this segment into $\mathcal{R}$; in contrast, if it is the right endpoint of a segment, it deletes this segment from $\mathcal{R}$. Note that all the segments intersecting with l_v are stored (in order from bottom to top) in $\mathcal{R}$. In particular, when l_v moves from one endpoint to another endpoint, it always checks whether or not newly adjacent segments intersect with each other; If so, it computes the intersection. In this way, all intersections can be obtained finally[4].

5.3. *Avoiding False Intersections and Speeding Up Lookups*

We can easily see that the plane sweep method directly inserts a segment into the balanced tree $\mathcal{R}$, if the point p ($\in \mathcal{Q}$) is the left endpoint of the segment. Instead, we assign two labels to the segment before it is inserted into $\mathcal{R}$. Note that the segment discussed here refers to the *processed related edge*. For clearness, we denote by lb_1 and lb_2 the two labels, respectively.

lb_1 is the boolean type, identifying that a segment is from which one of the two circular-arc polygons. Specifically, if the segment is from $\mathcal{P}_1$, we assign *true* to lb_1; otherwise, we assign *false* to lb_1. Recall that the plane sweep method always checks whether or not two segments intersect with each other, when they are adjacent. Our proposed method does not need to check them regardless of whether or not they intersect, if the first labels of two adjacent edges have the same value. This can avoid the unnecessary test and the "false" intersections.

lb_2 is an integer type denoting a serial number, which corresponds to the "id" of an item stored in the sequence list (note: the "id" information of each item is implied, as we store the items using the sequence list, i.e., array). When we detected an intersection, this label can help us quickly find the item in the sequence list, and then insert the intersection into this item. See Figure 3(b) for an example, lb_1 and lb_2 of edge $\overline{ab}$ are assigned to *true* and 1, respectively. When we detected the intersection i_1, we thus can quickly know that we should insert the intersection into $S_1[1]$ (i.e., the first item of S_1). Otherwise, we have to scan the sequence list in order to insert the intersection into an appropriate item, this way is inefficient especially when $|S_1|$ (or $|S_2|$) is large.

Algorithm 1 *ConstructSequenceLists*

Input: Circular-arc polygons $\mathcal{P}_1$ and $\mathcal{P}_2$
Output: Sequence lists S_1 and S_2, related edge sets R_1 and R_2
1: Find the MBRs, effective axis and extended boundary lines
2: **for** each $i \in \{1,2\}$ **do**
3: $R_i \leftarrow$ related edges from $\mathcal{P}_i$
4: Create two empty sequence lists S_1 and S_2
5: **for** each $i \in \{1,2\}$ **do**
6: *InitializeSequenceList* (R_i, S_i) // cf., Algorithm 2
7: Sort the endpoints of the segments (from S_1, S_2), and put them into the priority queue $\mathcal{Q}$
8: Initialize the empty balanced tree $\mathcal{R}$
9: **for** each point $p \in \mathcal{Q}$ **do**
10: Let s be the segment containing the point p, and t be the segment immediately above or below s
11: **if** (p is the left endpoint of segment s)
12: Assign two "labels" to s, and insert s into $\mathcal{R}$
13: **if** ($s.lb_1 \neq t.lb_1$)
14: **if** (s intersects with t)
15: Insert the intersection into $\mathcal{Q}$, and also insert into S_1 and S_2
16: **else if** (p is the right endpoint of segment s)
17: **if** ($s.lb_1 \neq t.lb_1$)
18: **if** (s intersects with t and this intersection $\notin \mathcal{Q}$)
19: Insert this intersection into $\mathcal{Q}$, and it into S_1 also insert and S_2, respectively; delete s from $\mathcal{R}$
20: **else** // p is an intersection of two segments, say s and t
21: Swap the position of s and t // assume s is above t
22: Let t_1 be the segment above s, and t_2 be the segment below t
23: **if** ($s.lb_1 \neq t_1.lb_1$ or $t.lb_1 \neq t_2.lb_1$)
24: **if** (s intersects with t_1, or t intersects with t_2)
25: Insert the intersection point into Q, and also insert it into S_1 and S_2, respectively
26: **return** S_1 and S_2, R_1 and R_2

5.4. *The Algorithm*

Let R_1 and R_2 be the related edges from $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Given a segment s, we use $s.lb_1$ and $s.lb_2$ to denote the two labels of segment s. Algorithm 1 illustrates the pseudo-codes of constructing the two sequence lists.

We first choose the *related edges* based on the extended boundary lines (Lines 1-3). Next, we construct two empty sequence lists and initialize them (Lines 4-6). After this, we compute the intersections (Lines 7-25). In particular, when we compute the intersections, two *labels* are assigned to the segment before it is inserted into the balanced tree (Line 12), and we use the two sequence lists to store the intersections (Lines 15, 19 and 25). Note that, the pseudo-codes of initializing the two sequence lists are listed in Algorithm 2. This algorithm decomposes non-x-monotone arcs, puts the *processed related edges* into the sequence lists in an orderly manner, and assigns appropriate values to the tri-value switches.

Lemma 5.1. *Given two circular-arc polygons with m and n edges, respectively, and assume there are l related edges between the two polygons, we have that constructing the two sequence lists can be finished in* $O(m+n+l+(l+k)\log l)$ *time, where k is the number of intersections.*

[4] Note that, in some cases the segments may be vertical line segments, or the segments may be tangent, or many segments possibly intersect at one point; for these degenerated cases, please refer to the papers (e.g., [2, 11, 15, 21, 23]) for more details. Unless stated otherwise, degenerated cases are processed using existing techniques and/or a straightforward adaptation from existing techniques. We no longer expatiate them separately for saving space (as they are tedious, and are not the focus of the paper).

Proof. To obtain the *related edges*, we first need to find the MBRs of two polygons, which takes linear time. We next determine the *effective axis* by comparing the horizontal and vertical spans of the intersection set of two MBRs, which can be finished in constant time. Furthermore, the *extended boundary lines* can be obtained in constant time once we obtain the effective axis. Based on two extended boundary lines, we finally obtain the *related edges* by comparing the geometrical relationship between each edge and extended boundary lines, which also takes linear time. Thus, Lines 1-3 take $O(m+n)$ time.

Creating two empty sequence lists takes constant time. In addition, in order to initialize the two sequence lists, we need to decompose each non-x-monotone arc. Decomposing a single arc can be finished in constant time. In the worst case, all the *related edges* are non-x-monotone arcs. Even so, there are no more than $3l$ items in the two sequence lists, according to Lemma 4.1. Hence initializing two sequence lists takes $O(l)$ time. Sorting all the endpoints of segments in the priority queue $\mathcal{Q}$ takes $O(l \log l)$ time, and initializing the balanced tree $\mathcal{R}$ takes constant time. Thus, Lines 4-8 take $O(l + l \log l)$ time.

As there are no more than $3l$ segments in S_1 and S_2, the number of endpoints thus is no more than $6l$. So we can easily know that the number of executions of the **for** loop (Line 9) is no more than $6l+k$. Within the **for** loop, each operation (e.g., insert, delete, swap, find the above/below segment) on $\mathcal{R}$ can be finished in $O(\log l)$ time, as the number of segments in $\mathcal{R}$ never exceeds $3l$. Additionally, each of other operations (e.g., assign labels to the segment, determine if two segments intersects with each other) can be finished in constant time. Thus, Lines 9-25 take $O((6l+k)\log l)$ time, i.e., $O((l+k)\log l)$ time. Putting it together, this completes the proof. □

Algorithm 2 *InitializeSequenceList*

```
Input: R_i, S_i
Output: S_i
temp ← 1 // the temp is used to set the tri-value switch
for each related edge r ∈ R_i do
    if (r is a non-x-monotone circular arc )
        Decompose it and put the decomposed arcs into S_i, and
         set the value of each tri-value switch to "temp"
        if (temp=1)
            temp ←2;
        else // temp=2
            temp ←1;
    else // r is not a non-x-monotone circular arc
        Put it into S_i, set the value of tri-value switch to "0"
```

We have shown how to construct two sequence lists. It is easy to know that we cannot get the resultant polygon based on *only* the information stored in the two sequence lists. Next, we are ready to merge the information in them and part of information in the original linked lists, and store the 'merged information' using two *new linked lists*. For ease of exposition, we call this step 'building two new linked lists'. Note that the two new linked lists will be significantly used in Section 7, as we need to traverse them to get the resultant polygon.

6. Building Two New Linked Lists

To construct two new linked lists, on the whole we first initialize two (empty) new linked lists, and then copy the information from the *original linked lists* to the *new linked lists* while we replace those *related edges* using the information stored in the two sequence lists. Note that there are two important yet easy-to-ignore issues needed to be handled when we construct new linked lists. We next check these issues, and then present the algorithm of constructing new linked lists.

6.1. Eliminating The Ambiguity

Recall Section 4.1, we always add an *appendix point* between two vertices if an edge is a circular arc. When we replace *related edges* with the information stored in sequence lists, we also have to ensure this property. It is easy to know that, when the intersections appear on a circular arc, this arc will be decomposed by these intersections. We thus have to add the new appendix point for each sub-segment, in order to eliminate the ambiguity.

Lemma 6.1. *Suppose there are k intersections on a circular arc; then, we need to insert at least k and at most $k+1$ new appendix points.*

Proof. Since k intersections can subdivide a complete circular arc into $k+1$ small circular arcs, and for each small circular arc one *appendix point* is needed and enough to eliminate the ambiguity. Clearly, $k+1$ *appendix points* are needed for $k+1$ small circular arcs. Note that, there is an *appendix point* beforehand. Therefore, when there is no any intersection (among all these intersections) that is *coinciding with* this existing *appendix point*, only k new *appendix points* are needed. Otherwise, we need $k+1$ new *appendix points*. □

Besides the above issue, another issue is to handle the decomposed arcs. We decomposed non-x-monotone arcs into x-monotone arcs ever, we thus need to merge them. The natural solution is to compare each pair of adjacent edges of the resultant polygon, checking if they can be merged into a single arc. This way however is inefficient because (i) most of edges of the resultant polygon may not need to be merged; and (ii) given two adjacent arcs, let C_1 and C_2 respectively denote their corresponding circles; checking if the two arcs can be merged into a single arc needs to compute the centres of C_1 and C_2, this will use trigonometric functions (which could have been avoided). We next show how to efficiently merge them, with the help of the *tri-value switch* (recall Section 5.2).

6.2. Efficiently Merging Decomposed Arcs

We merge the decomposed arcs when constructing new linked lists, rather than merge them after obtaining the resultant polygon. In particular, we here utilize the information stored in the tri-value switch to improve the efficiency. Specifically, given an item $S_i[j]$, if $S_i[j].c = 1$ (or 2), we continue to fetch its next item $S_i[j+1]$ from the sequence list if $S_i[j].c = S_i[j+1].c$. In this way, a group of consecutive items are fetched from the sequence list. W.l.o.g, assume that we have fetched λ consecutive items, $S_i[j], \cdots, S_i[j+\lambda-1]$. Then, we do as follows.

- If $S_i[j].b = S_i[j+1].b = \cdots = S_i[j+\lambda-1].b = \emptyset$, we discard the fetched items instead of merging them. This is because there is no intersection on these decomposed arcs, the merged result should be the same as the edge in the original linked list.
- Otherwise, we insert new appendix points, merge decomposed arcs, and replace the edge in the original linked list.

Let us revisit Figure 5(b); recall that there are eight items in S_1, and the values in their tri-value switches are "0, 1, 1, 2, 2, 1, 1, 0", respectively. Although $S_1[2].c = S_1[3].c = 1$, we discard the two items instead of merging them, as $S_1[2].b = S_1[3].b = \emptyset$. Similarly, we also discard the items $S_1[4]$ and $S_1[5]$. Note that, for the 6th and 7th items, $S_1[6].c = S_1[7].c = 1$ and $S_1[7].b \neq \emptyset$; thus, we insert a new appendix point, merge the two decomposed arcs, and use the merged result to replace the edge in the original linked list.

Note that, the consecutive items mentioned earlier are actually the decomposed arcs generated from a single non-x-monotone circular arc. According to Lemma 4.1, we can easily obtain the following corollary.

Corollary 6.1. *Let λ be the number of consecutive items, we have that $\lambda \leq 3$ and $\lambda \geq 2$.*

Algorithm 3 *BuildNewLinkedLists*

Input: $\mathcal{P}_1$ and $\mathcal{P}_2$, S_1 and S_2, R_1 and R_2
Output: $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$

```
1:  Set P1* = P2* = ∅, and j ← 1
2:  for each i ∈ {1,2} do
3:    for each edge e ∈ Pi do
4:      if (e ∉ Ri )
5:        Add e to Pi*
6:      else // e is a related edge
7:        if (si[j].c = 0) // not a decomposed arc
8:          if (Si[j].b = ∅) // no intersection
9:            j ← j+1, and add e to Pi*
10:         else // Si[j].b ≠ ∅
11:           if (Si[j] is a circular arc)
12:             Insert new appendix points
13:           Put the information from Si[j] into Pi*, and set j ← j+1
14:       else // si[j].c = 1 (or 2)
15:         Set tri = Si[j].c, and λ ← 0
16:         do // copy the consecutive decomposed arcs
17:           λ ← λ+1, temp[λ] ← Si[j], j ← j+1
18:         while Si[j].c = tri
19:         if (temp[1].b = ··· = temp[λ].b = ∅)
20:           Put e into Pi*
21:         else
22:           Insert new appendix points, merge decomposed arcs, and
              put the merged result into Pi*
23: return P1* and P2*
```

6.3. The Algorithm

Let $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ be the two new linked lists, respectively. Algorithm 3 depicts the pseudo-codes of constructing two new linked lists. For each edge e in the *original linked list*, we check whether it is a *related edge*. If so, we further check whether $S_i[j]$ is a decomposed arc. Lines 7-13 are used to process the case when it is not a decomposed arc. In contrast, Lines 14-22 are used to handle the opposite case. In this case, we first fetch all the consecutive decomposed arcs (Lines 15-18), and then check if there are intersections on these decomposed arcs. If it is not, we put the edge e into $\mathcal{P}_i^*$ (Lines 19-20). Otherwise, we insert new appendix points, merge decomposed arcs, and put the merged result (instead of e) into $\mathcal{P}_i^*$ (Lines 21-22).

Lemma 6.2. *Suppose we have obtained the two sequence lists S_1 and S_2; then, constructing the two new linked lists $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ takes $O(m+n+l+k)$ time.*

Proof. Inserting a single *appendix point* takes constant time. In the worst case, all the intersections are located on arcs rather than on line segments. Even so, there are no more than $2k$ new *appendix points* according to Lemma 6.1. Hence inserting *appendix points* takes $O(k)$ time. Merging λ consecutive decomposed arcs takes constant time, as $\lambda \leq 3$ (cf., Corollary 6.1). In the worst case, all the related edges are non-x-monotone arcs, hence merging all consecutive decomposed arcs takes $O(l)$ time.

Since the number of edges in $\mathcal{P}_1$ and $\mathcal{P}_2$ is $m+n$, the number of executions of the second **for** loop (Line 3) is $m+n$. Specifically, the number of executions of Line 4 is $m+n-l$, and the one of Line 6 is l. Even if all related edges are non-x-monotone arcs, the number of executions of Line 17 is no more than $3l$. Furthermore, within the **for** loop, each operation takes constant time (note: here we no longer consider the time for inserting new appendix points and merging decomposed arcs, as we have analysed them in the previous paragraph). Therefore, Lines 4-5 and Lines 7-22 take $O(m+n-l)$ and $O(3l)$ time, respectively. Putting it all together, this completes the proof. □

7. Traversing

In the previous section, we have obtained $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$. This section shows in detail how to get the resultant polygon by traversing them. In order to correctly traverse $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$, we need to assign the *entry-exit* properties to intersections.

7.1. Entry-Exit Property

The entry-exit property is an important symbol that was ever used in many papers focusing on boolean operation of traditional polygons (see e.g., [11, 15]). The followings show this technique can be used to the case of our concern as well. Specifically, we assign the intersections with the *entry* or *exit* property *alternately*. Note that the *entry-exit* property for the first intersection in $\mathcal{P}_1^*$ ($\mathcal{P}_2^*$) is determined as follows. W.l.o.g, assume the first intersection is i (i') in $\mathcal{P}_1^*$ ($\mathcal{P}_2^*$), and the previous node of i (i') is $i.prev$ ($i'.prev$). We check if $i.pre$ ($i'.pre$) is outside the input polygon $\mathcal{P}_2$ ($\mathcal{P}_1$). If so, we assign the *entry* (*exit*)

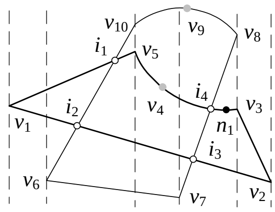


Figure 6: Example of assigning entry-exit properties. $\mathcal{P}_1 = \{v_1, v_2, v_3, \widetilde{v_4}, v_5\}$ and $\mathcal{P}_2 = \{v_6, v_7, v_8, \widetilde{v_9}, v_{10}\}$.

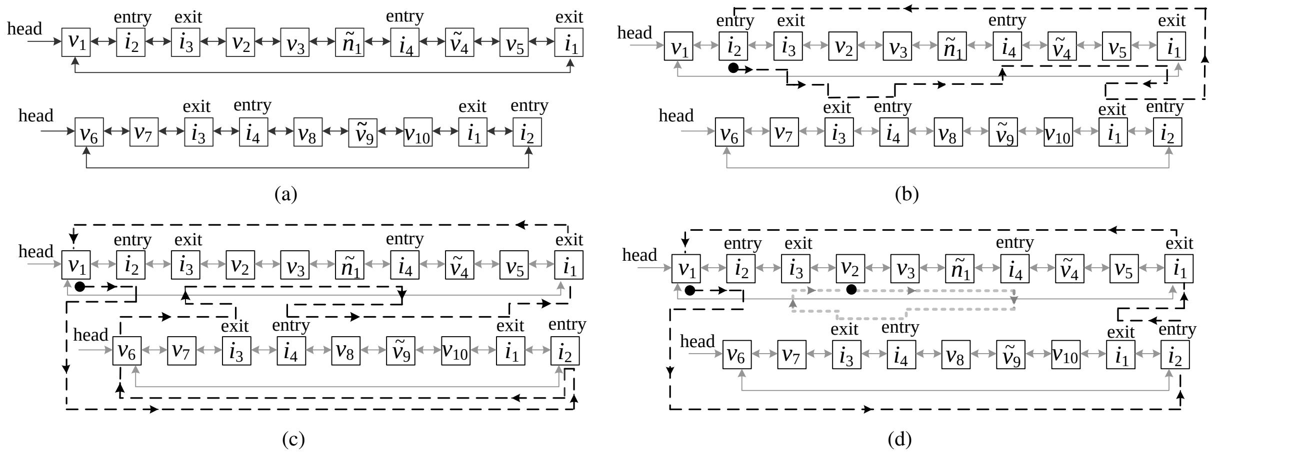


Figure 7: Illustrations of entry-exit properties and traversing rules. (a) Entry-exit properties of $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$. (b) Intersection operation. (c) Union operation. (d) Difference operation.

property to i (i'). See Figure 6 for an example. $\mathcal{P}_1$ and $\mathcal{P}_2$ are two input polygons. We can easily obtain the two new linked lists $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ using algorithms discussed before. Here $\mathcal{P}_1^* = \{v_1, i_2, i_3, v_2, v_3, \widetilde{n_1}, i_4, \widetilde{v_4}, v_5, i_1\}$, and $\mathcal{P}_2^* = \{v_6, v_7, i_3, i_4, v_8, \widetilde{v_9}, v_{10}, i_1, i_2\}$. Regarding to $\mathcal{P}_1^*$, since the previous node of i_2 is v_1 which is outside $\mathcal{P}_2$, we assign "*entry*, *exit*, *entry*, *exit*" to "i_2, i_3, i_4, i_1", respectively. The *entry-exit* properties of "i_2, i_3, i_4, i_1" in $\mathcal{P}_2^*$ are obtained similarly. See Figure 7(a).

Once the *entry-exit* properties are assigned to intersections, we then obtain the resultant polygon based on the traversing rules below.

7.2. Traversing Rules

Let i_s be an intersection (point) of $\mathcal{P}_1^*$ such that i_s has the *entry* property. Let v_s be a vertex of $\mathcal{P}_1^*$ such that v_s does not locate in $\mathcal{P}_2^*$. There are three typical boolean operations: intersection, union and difference. Note that in the rest of discussion, the default traversing direction is counter-clockwise, unless stated otherwise.

Intersection. We start to traverse $\mathcal{P}_1^*$ using i_s as the starting point. Once we meet an intersection with the *exit* property, we shift to $\mathcal{P}_2^*$, and traverse it. Similarly, if we meet an intersection with the *entry* property in $\mathcal{P}_2^*$, we shift back to $\mathcal{P}_1^*$. In this way, a circuit will be produced. After this, we check if there is another intersection of $\mathcal{P}_1^*$ such that (i) it has the *entry* property; and (ii) it is not a vertex of the produced circuit. If no such an intersection, we terminate the traversal, and this circuit is the intersection between $\mathcal{P}_1$ and $\mathcal{P}_2$. Otherwise, we let this intersection as a new starting point, and traverse the two new linked lists (using the same method discussed just now), until no such an intersection exists. In the end, we get multiple circuits, which are the intersection between $\mathcal{P}_1$ and $\mathcal{P}_2$. See Figure 7(b) for an example, we first choose i_2 in $\mathcal{P}_1^*$ as the starting point, and then traverse the two linked lists, getting a circuit (see the dashed lines). Moreover, there is no other intersection satisfying the two conditions mentioned before. Therefore, the intersection is $\{i_2, i_3, i_4, \widetilde{v_4}, v_5, i_1\}$.

Union. For this case, we, however, traverse $\mathcal{P}_1^*$ using v_s as the starting point. Once we meet an intersection with the *entry* property, we shift to $\mathcal{P}_2^*$, and traverse it. Similarly, if we meet an intersection with the *exit* property in $\mathcal{P}_2^*$, we shift back to $\mathcal{P}_1^*$. In this way, a circuit will be produced, which is the union between $\mathcal{P}_1$ and $\mathcal{P}_2$. See Figure 7(c), for example, we first choose v_1 as the starting point, and then traverse the two linked lists, until we are back to the starting point v_1. Therefore, $\{v_1, i_2, v_6, v_7, i_3, v_2, v_3, \widetilde{n_1}, i_4, v_8, \widetilde{v_9}, v_{10}, i_1\}$ is the union between $\mathcal{P}_1$ and $\mathcal{P}_2$.

Difference. The first several steps are the same as the ones in the *union* operation, *but* we traverse $\mathcal{P}_2^*$ in <u>clockwise</u> direction. Similarly, if we meet an intersection with the *exit* property in $\mathcal{P}_2^*$, we shift back to $\mathcal{P}_1^*$. In this way, a circuit will be produced. Furthermore, we check if there is another vertex of $\mathcal{P}_1^*$ such that (i) it is not a vertex of any produced circuit; and (ii) it does not locate in $\mathcal{P}_2^*$. If no such a vertex, we terminate the traversal, and this circuit is the difference between $\mathcal{P}_1$ and $\mathcal{P}_2$. Otherwise, we let the vertex as a new starting point, and traverse the two new linked lists (using the same method discussed just now), until no such a vertex exists. In the end, we get multiple circuits, which are the difference between $\mathcal{P}_1$ and $\mathcal{P}_2$. See Figure 7(d), for example, we shall get that the difference between $\mathcal{P}_1$ and $\mathcal{P}_2$ consists of two parts, i.e., $\{v_1, i_2, i_1\}$ and $\{v_2, v_3, \widetilde{n_1}, i_4, i_3\}$.

7.3. The Algorithm

In some cases the result consists of multiple circuits, we denote by l_j the linked list used to store the jth circuit. Let n_d be a node of $\mathcal{P}_i^*$ where $i \in \{1,2\}$, we denote by c_n the current node when we traverse $\mathcal{P}_i^*$, and denote by $c_n.next$ the next node of c_n. Furthermore, we denote by i_s' the intersection of $\mathcal{P}_1^*$ such that i_s' has the entry property and it is not a vertex of any produced circuit, and we use $\exists(i_s') = true$ to denote that there exists such a point. Algorithm 4 illustrates the pseudo-codes of obtaining the intersection result (remark: the ones of other two operations can be constructed similarly by traversing rules, omitted).

Lemma 7.1. *Given the two new linked lists $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$, to obtain the resultant polygon takes $O(k+m+n+l)$ time.*

Proof. Assigning the *entry-exit* property to each intersection takes constant time, and there are k intersections on each new linked list. Thus, assigning *entry-exit* properties to intersections takes $O(k)$ time.

$\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ are used to generate the resultant polygon, they store the vertices, *appendix* points, and intersections. The number of vertices is $2(m+n)$. In the worst case, all edges of two input polygons are circular arc segments, implying that the number of *appendix* points in the input polygons is $m+n$; in this case, all the k intersections are located on arcs, we need to insert new *appendix* points, and the number of new *appendix* points is no more than $k+1$, according to Lemma 6.1. So the number of all *appendix* points in $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ is no more than $m+n+k+1$. Therefore, the total number of nodes in $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ is no more than $3(m+n)+2k+1$. Further, each operation on a node (e.g., determine the type of a node, insert a node into the resultant polygon) takes constant time. Therefore, the traversal takes $O(3(m+n)+2k+1)$ time. Putting it all together, we have that obtaining the resultant polygon takes $O(m+n+k+l)$ time when $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ are given beforehand[5]. □

Algorithm 4 *TraverseLinkedLists*

Input: $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$
Output: $\mathcal{P}_3$
1: Set $j=1$
2: **for** each $i \in \{1,2\}$ **do**
3: Assign entry-exit property to $\mathcal{P}_i^*$
4: **do**
5: **if** (j=1)
6: Choose a starting point i_s from $\mathcal{P}_1^*$
7: **else**
8: Let $i_s \leftarrow i'_s$ // i.e., let i'_s be a new starting point
9: Set $c_n \leftarrow i_s$, and $l_j = \emptyset$
10: **do**
11: Put c_n into l_j
12: **if** (c_n.*next* is not an intersection point)
13: Let $c_n \leftarrow c_n$.*next*, and put c_n into l_j
14: **else**
15: Shift to $\mathcal{P}_2^*$, choose the node n_d such that $n_d = c_n$.*next*, let $c_n \leftarrow n_d$, and put c_n into l_j
16: **if** (c_n.*next* is not an intersection point)
17: $c_n \leftarrow c_n$.*next*, and put c_n into l_j
18: **else**
19: Shift to $\mathcal{P}_1^*$, choose the node n_d such that $n_d = c_n$.*next*, and let $c_n \leftarrow n_d$
20: **while** ($c_n \neq i_s$)
21: Let $\mathcal{P}_3 \leftarrow \mathcal{P}_3 \cup l_j$, and set $j \leftarrow j+1$
22: **while** ($\exists(i'_s) = true$)
23: **return** $\mathcal{P}_3$

Up to now, we have addressed all the main steps of our algorithm — RE2L. We next analyse its complexity.

8. Time/Space Complexity

We analyse the complexity of our algorithm *using* the intersection operation as a sample (note: the complexity of other two operations is the same as the one of this operation, and can be derived similarly, omitted).

Theorem 8.1. *Given two circular-arc polygons with m and n edges, respectively, and assume there are l related edges between the two circular-arc polygons. Then, to achieve boolean operation on them takes $O(m+n+(l+k)\log l)$ time, using $O(m+n+k)$ space, where k is the number of intersections.*

Proof. Our algorithm consists three main steps, and they take time $O(m+n+l+(l+k)\log l)$, $O(m+n+k+l)$, and $O(m+n+k+l)$, respectively (see Lemmas 5.1, 6.2 and 7.1). Putting these results together, hence the time complexity is $O(m+n+(l+k)\log l)$.

The space used in our algorithm mainly consists of two groups of related edges R_1 and R_2, two sequence lists S_1 and S_2, the priority queue $\mathcal{Q}$, two new linked lists $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$, and the balanced tree $\mathcal{R}$. (Remark: the input polygons $\mathcal{P}_1$, $\mathcal{P}_2$, and the output polygon $\mathcal{P}_3$ are the input and output data; by the convention[6], we do not need to consider them when we analyse the space complexity.)

Specifically, R_1 and R_2 have the size of $O(l)$, as they are used to store the *related edges*. S_1 and S_2 are used to store the *processed related edges*. In the worst case, the number of *processed related edges* is no more than $3l$, according to Corollary 5.1. So S_1 and S_2 have the size of $O(3l)$. Recall Algorithm 1, the priority queue $\mathcal{Q}$ is used to store the endpoints of *processed related edges* and the intersections, and so $\mathcal{Q}$ has the size of $O(6l+k)$. Regarding to the balanced tree $\mathcal{R}$, it has the size of $O(l)$ at most, as it is used to store the segments *currently* intersecting the sweep line. Furthermore, $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$ have the size of $O(3(m+n)+2k+1)$, see the proof of Lemma 7.1. Putting it all together, we have that the space complexity of our algorithm is $O(m+n+k+l)$. In addition, the upper bound of the parameter l is $m+n$. This completes the proof. □

We can see that our algorithm roughly consumes linear space when k is small. The running time also approximates to linear complexity when l and k are small. It is noteworthy that a straightforward adaptation from the plane sweep algorithm (see e.g., the 'Standard' method described in Section 10.1) requires $O((m+n+k)\log(m+n))$ time. In other words, even if k is small (e.g., $k << m+n$), it is also *linearithmic time*. We remark that, for boolean operations on circular-arc polygons, the $O((m+n+k)\log(m+n))$ result is actually the state-of-the-art competitor in terms of computational complexity.

Although our algorithm has some advantages to some extent, we have to point out that in the worst case (note: this case is possible although it is not the usual case), i.e., $l=m+n$, the running time deteriorates to $O((m+n+k)\log(m+n))$, which is equal to the one of the standard method. In this case, the advantages of the proposed algorithm disappear, viewed from the theoretical perspective. To this step, an interesting issue is: when $l=m+n$, whether or not its practical efficiency is also the same as the one of the standard method? We will experimentally evaluate our algorithm as well as the competitors in Section 10, after we introduce some immediate extensions.

[5] It is simple to determine i'_s (cf., Lines 8 and 22). We just need to collect all the intersections with entry properties in a data structure when we assign entry-exit properties to intersections, and then remove the intersections from this data structure once they have already been visited in the process of traversing. After a circuit is formed, we check if this data structure is empty. If otherwise, any intersection stored in this data structure can be taken as i'_s.

[6] As an example, the *bubble sort algorithm* takes $O(1)$ space for sorting arbitrary n natural numbers.

9. Extensions

While this paper focuses on boolean operation of circular-arc polygons, our techniques can be easily extended to compute boolean operation of traditional polygons. Assume there are two traditional polygons, for example, we can also use two lists to represent them. In this case, the Tag domain is unneeded as the traditional polygons do not need the appendix points. We can also choose *related edges* based on the extended boundary lines, and store them using *two sequence lists*. Note that the third domain of the sequence list is unneeded, as here all related edges are straight line segments. When computing the intersections, we can also use *two labels* to speed up the process of inserting the intersections into their corresponding edges, and to avoid *false* intersections. Next, we also construct *two new linked lists*, by using the information stored in two sequence lists to replace the related edges in the original linked lists. Particularly, we here do not need to insert new appendix points and merge the decomposed arcs, as the traditional polygons have no such information. We finally get the resultant polygon by *traversing* the two new linked lists, it is the same as that in Section 7.

Furthermore, the discussions presented in previous sections assumed the circular-arc polygons to be operated have no holes. If we want to handle the opposite case, this can be easily achieved by a straightforward adaptation of our proposed method. Assume we want to compute the intersection of two circular-arc polygons with holes, for example, we can use multiple lists to represent the circular-arc polygon with holes. One is to represent the outer boundaries of the circular-arc polygon, others are respectively to represent the boundaries of each hole. We can first compute the intersection of the outer boundaries of two polygons, and then use this intersection result to subtract each hole of the two polygons. All of these steps are quite straightforward, when our proposed method is given beforehand.

Finally it is also immediate to compute boolean operation on circular-arc polygons with *self-intersection*. Assume we want to compute the intersection, for example, we just need to make a minor modification on the traversing rule presented in Section 7.2. We also start to traverse $\mathcal{P}_1^*$ using an intersection with the *entry* property as the starting point, and shift to $\mathcal{P}_2^*$ when the number of intersections we meet in $\mathcal{P}_1^*$ is even. The main difference is that the *traversing direction* here is not always constant. To determine whether or not the traversing direction is needed to change when we shift from $\mathcal{P}_1^*$ ($\mathcal{P}_2^*$) to $\mathcal{P}_2^*$ ($\mathcal{P}_1^*$), a key step is to check if the *entry-exit* property of this intersection in $\mathcal{P}_1^*$ is different from the one in $\mathcal{P}_2^*$. If so, we need to change the traversing direction. Otherwise, we needn't.

10. Performance Evaluation

This section evaluates our algorithm experimentally. Specifically, Section 10.1 describes the baselines. The experimental settings and datasets are introduced in Section 10.2, and the results are reported in Section 10.3. We also investigate our proposed algorithm based on a specific application (Section 10.4).

Polygon	Coordinates
$\mathcal{P}_1$	(10,10), (40,10), (40,30), (<u>32.5,40</u>), (20,40),(15,30), (<u>25,22.5</u>),(15,15)
$\mathcal{P}_2$	(20,20), (<u>32.5,25</u>) (45,20), , (55,30), (<u>47,38.625</u>), (50,50), (30,45)

Table 1: The coordinates of vertices are listed counter-clockwise, and the left-bottom vertex is listed at first. The values tagged with "_" denote the coordinates of *appendix points*.

10.1. Methodologies

We compare our method (i.e., RE2L) with several baselines, which are either the algorithms used to handle more general case of polygons, or the simpler versions of our proposed method. We shortly introduce them as follows.

CGAL. We directly use the implementation of CGAL. The essence of this method is to directly invoke the algorithm of boolean operation on *general polygons*, defined as `GeneralPolygon_2` in CGAL. Specifically, the "CGAL::Cartesian<Number_type>" is used as the kernel, in which "Number_type" denotes the exact rational number-type by default (see the header file "arr_rational_nt.h" in CGAL for reference). Based on this kernel, we construct "CGAL::Gps_circle_segment_traits_2" trait class, and the following objects "Curve_2, X_monotone_curve_2, General_ polygon_2, Point_2" are used, which inherit the trait class above.

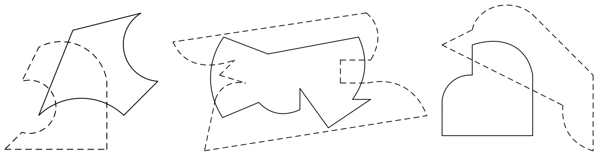
Berberich. We directly use the algorithm in [3], which is initially developed for computing boolean operations of conic polygons. This method employs the DCEL structure to represent the polygons. It first decomposes non-x-monotone conic curves and then computes the arrangement of segments using the plane sweep method; next, it uses the results of arrangement to compute the *overlap* of two polygons in order to achieve boolean operation. This algorithm is similar to that of CGAL, recall Section 2. (Remark: more information about the DCEL structure and computing the overlap of two polygons can be found in [5].)

Naïve. It is one of simpler versions of our proposed method. This method employs our proposed data structure, it however computes the intersections by comparing each pair of edges. Specifically, for each edge e of $\mathcal{P}_1$, it checks whether e intersects with the edges of $\mathcal{P}_2$. If so, it computes the intersections and inserts them into corresponding edges. It does in this way, until all edges of $\mathcal{P}_1$ are processed. The rest of steps are to assign the *entry-exit* properties and to traverse, which are the same as the ones of our proposed method. Note that Greiner-Hormann's algorithm [11], that was initially developed for boolean operations of traditional polygons, also computes the intersections by comparing each pair of edges, and the rest of steps also include traversing. In this regard, the Naïve method can be also looked as a generalization of Greiner-Hormann's method.

Standard. It is also a simpler version of our proposed method, but it is different from the previous one. Specifically, it employs the standard plane sweep algorithm to compute the intersections instead of checking each pair of edges. Note that it does not adopt the proposed optimization strategies (e.g., 'using related edges', 'avoiding false intersections', 'speeding up

Polygon	Coordinates
$\mathcal{P}_1$	(15,5),(88.5,16.5),(80.95,25.9),(69,27.5),(60,27.5),(60,35),(70,35),(72.35,43.05),(68.5,50.5),(4.5,40.5),(13.65,33.65),(25,35),(20,30),(28.5,27.5),(22.4,26.75),(18.5,22)
$\mathcal{P}_2$	(33.5,21),(39.5,17.15), (46.5,18.5),(46.5,25.5),(56,12.5),(70,22),(64,22),(67.9375,31.9643), (66,42.5),(36,37),(21.5,42.5),(17.4643,29.32145), (21,16)

Table 2: The coordinates of vertices are listed counter-clockwise, and the left-bottom vertex is listed at first.



(a) (b) (c)

Figure 8: The use cases for our experiments. The polygon with dashed lines is $\mathcal{P}_1$, and another one is $\mathcal{P}_2$. (a) For the first set of experiments. (b) For the fourth set of experiments. (c) A sampled example for which all algorithms work well, even if we use floating point data type.

the lookups', 'speeding up the merging of decomposed arcs'), others are the similar as the RE2L method. (Remark: the idea of the Standard method is roughly same to that of *CGAL* and *Berberich*, but it simply gets rid of the generality and employs a targeted data structure.)

10.2. Experimental Settings & Datasets

10.2.1. Settings

All the algorithms are implemented in C++ language, the versions of LEDA, CGAL and BOOST library are 6.2, 4.3, and 1.46.1, respectively. The proposed algorithm and its simpler versions do not employ other libraries except the *standard template library* (STL). The priority queue and the balanced tree mentioned in previous sections are implemented using a heap and a red-black tree, respectively. The experiments are conducted on a computer with 2.16GHz dual core CPU and 1.86GB of memory. By convention, we use the execution time to measure the efficiency. In our experiments, we run 100 times by default for each algorithm and then compute the average running time.

10.2.2. Datasets

Experiment 1. We manually produce two circular-arc polygons, each of them is less than 10 edges, for simplicity and for ease of reproducing the findings. The vertex coordinates of the two polygons (cf., Figure 8(a)) are listed in Table 1.

Experiment 2. To study the overall performance of these algorithms, we adopt thousands of circular-arc polygons. Specifically, given an integer n, a pair of circular-arc polygons with n edges are generated at random[7], and then each algorithm is executed alternately, using the pair of polygons as the input. This is done one thousand times. In each trial, the running time of each algorithm is recorded, and accumulated to previous trials. We compute the average value for estimating the average running time of each algorithm. Furthermore, we vary the value of n, and obtain the running time of each algorithm using the same method mentioned above.

Experiment 3. We use the parameter "`double`" to replace the parameter "Number_type" in the kernel "CGAL::Cartesian <Number_type>" (the 1st approach), and also the parameter "Rational" in the kernel "CGAL::Cartesian <Rational>" (the 2nd approach), in order to investigate the performance when all these algorithms use the floating point number-type. We randomly generate pairwise circular-arc polygons as the test data.

Experiment 4. To answer the interesting issue mentioned in Section 8, we use the polygons satisfying the condition $l = m + n$ as the input, see Figure 8(b) (it is a sample). The vertex coordinates of these two polygons are listed in Table 2.

Experiment 5. Inspired by the curiosity, we also investigate the scalability of our proposed algorithm using polygons with a larger number of edges. Note that in this set of experiments almost all the polygons generated are self-intersection polygons when the number of edges is equal to or larger than 100. This is because it is pretty difficult to generate polygons without self-intersections when the number of edges is large, using the method mentioned in Footnote 7. Specifically, in this case we do not employ the constraint (ii), see Footnote 7.

Method	CGAL	Berberich	Naive	Standard	RE2L
Time (sec.)	0.0273	0.0287	0.00239	0.00175	0.00112
Impr. fac.	24.375	25.625	2.13	1.5625	—

Table 3: The average running time in the first set of experiments.

10.3. Experimental Results

10.3.1. Results of The First Experiment

Table 3 lists the results of Experiment 1. Specifically, the methods are listed in Row one, the average running time of each method is listed in Row two, and the improvement factors[8] of our algorithm over the baseline methods are shown in Row 3. From this table, we can see that the proposed method outperforms its simpler versions, demonstrating the effectiveness of our proposed strategies. Interestingly, the simpler versions of the proposed method yet outperforms the former two methods, let alone the proposed method. To some extent this verifies our previous claim — directly executing existing algorithms used

[7]Generally speaking, we first randomly generate two rectangles such that they are overlapping each other. Then, we randomly generate n points in *each* rectangle one by one such that they satisfy two constraints: (i) the segment joining the jth point and the $(j-1)$th one cannot intersect with any other segment except the segment joining the $(j-1)$th point and the $(j-2)$th one, where $j \geq 4$; and (ii) the segment joining the 1st point and the nth one cannot intersect with any other segments except its two adjacent segments. These n points will be used as the vertices of the circular-arc polygon. Finally, we import circular-arc segments by inserting a set of *appendix points*.

[8]Here the improvement factor refers to the ratio of time. Assume that the exectution time of the 'Standard' method is 0.2 seconds, the one of the proposed method is 0.05 seconds, for example, the improvement factor is $\frac{0.2}{0.05} = 4$.

n	CGAL	Berberich	Naive	Standard	RE2L
5	0.0281	0.0293	0.00234	0.0018	0.00107
10	0.0609	0.0772	0.0062	0.0041	0.00256
20	0.1282	0.1297	0.0125	0.0072	0.00369
30	0.1656	0.1741	0.0328	0.0176	0.00614
40	0.5172	0.5672	0.0391	0.0182	0.00683
50	0.61	0.681	0.0594	0.0213	0.00851

Table 4: The average running time of each algorithm, where n denotes the number of edges of each polygon.

Method	CGAL	Berberich	Naive	Standard	RE2L
Time (sec.)	0.0112	0.0127	0.00192	0.00151	0.000948
Imp. fac.	11.814	13.396	2.025	1.593	—

Table 5: The average running time when machine floating type is used for all these methods.

to compute boolean operation of conic and/or general polygons is usually not efficient enough.

Compared to the former two methods, although the latter three ones adopt a different data structure, but we note that the 'Standard' method also directly employs the plane sweep method, similar to the former two ones. Viewed from the theoretical perspective, the former two methods should have the performance similar to the "Standard" method. To further verify this phenomenon and explain it, we hence conduct another set of experiments, evaluating the overall performance based on larger datasets.

10.3.2. Larger Datasets

Table 4 lists the detailed results of Experiment 2. The results show that the proposed method outperforms other four ones as well, and is several orders of magnitude faster than the former two ones. By comparing the differences of these methods, we can easily see that the reason for the larger running time of the former two methods may well be that both methods employ the CGAL library and the DCEL structure[9]. Even so, it is noteworthy that comparing the former two methods with the latter three ones might be not very fair, as the latter three ones use *floating point arithmetic* (similar to that in [11, 14–18, 20–22, 25]), whereas the former two ones use *exact algebraic arithmetic*, which is more robust.

To make a more fair comparison, a simple remedy is to let the input data type be also *machine floating point* for the former two ones. In the next paragraph, we report our findings when all these methods use machine floating point type.

10.3.3. Floating Point Number-type

Specifically, we use `double` type as the input data type for all these methods, recall Section 10.2.2. After this, we generate randomly a pair of circular-arc polygons and then attempt to call these methods. Unfortunately, the former two methods fail with the message 'precondition violation' for many inputs. To this step, we also attempt to generate many other (pairs of) circular-arc polygons, and to test them. As a result, in most of cases the runtime exceptions are also reported. We also realize that, for a few test data (i.e., circular-arc polygons generated randomly), all these methods can work correctly[10]. For example, when the vertex coordinates of two input circular-arc polygons (cf., Figure 8(c)) are {(5,2), (5.125,1), (6,0.5), (6,3), (4,5), (2.875,5.25), (2,4.5), (1,4)} and {(2,4), (2,3), (1.25,2.75), (1,2), (1,1), (4,1), (4,3), (3.25,4)}, all these methods can work correctly. We remember these two polygons and run 100 times for each method (using these two polygons as the input). Table 5 depicts their average running time. This table shows us that the proposed method also outperforms its simpler versions. Again, the simpler versions of the proposed method also outperforms the former two methods, which is similar to our previous findings (although the improvement factors decrease in terms of the former two methods). This verifies our original claim in a more justified manner.

Method	CGAL	Berberich	Naive	Standard	RE2L
Time (sec.)	0.07844	0.08023	0.00718	0.004652	0.002965
Imp. fac.	26.4553	27.058	2.4215	1.5689	—

Table 6: The average running time when the two polygons satisfy the condition $l = m + n$.

10.3.4. The Case "$l = m + n$"

Table 6 reports the results of Experiment 4. Interestingly, the proposed method still outperforms the 'Standard' method[11] (note: both algorithms have the same time complexity in this case, recall Section 8). This phenomenon reveals that (i) two algorithms with the same time complexity might have different performance results in terms of execution time[12]; (ii) here other heuristics or optimization strategies (except the heuristic "utilizing related edges") also bring us the benefits; and (iii) the performance gain of other optimization strategies is larger than the cost consumed by the operator "choosing related edges".

10.3.5. Scalability

Figure 9 reports the results when we vary the number of edges (of polygons) from 5 to 500. The columns denote the numbers of intersections and related edges respectively, whereas the curves denote the running time. From this figure, we can see that the RE2L has better scalability compared to its competitors, as the growth speed of the running time is slower than

[9] We remark that computing the intersections of two polygons is unavoidable for any clipping algorithm, and it is a dominant step [5, 8, 11]. Both the former two methods and the "Standard" method adopt the plane sweep method to compute the intersections, their performance differences however are so great. This reminds us that the gap may well be due to the usage of the CGAL library and the DCEL structure (the former might be the major reason).

[10] The former two methods are originally designed for the exact algebraic arithmetic, here we use the machine floating point as the input data type; this could be the reason why most of cases cannot work correctly.

[11] We remark that the results are similar when the input polygons with more edges are used, omitted for saving space.

[12] This argument could be another reason why the performance of the'Standard' method is significantly different from the former two methods. That is, it is possible that the CGAL library used in the former two methods pays more attention to the robustness and stability, at the expense of the part of the performance. The further explanation is beyond the topic of this paper.

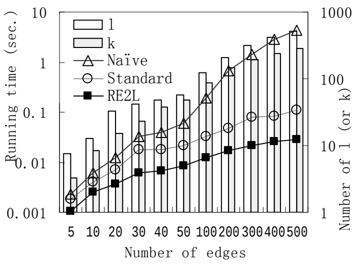
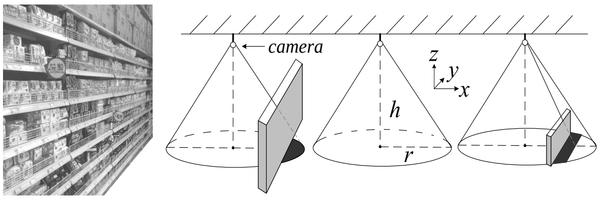


Figure 9: The experimental results when we use larger data sets.



(a) (b)

Figure 10: (a) A corner of the 'Auchan supermart'. (b) An illustration of the visual range.

Obstacle							
Height	4.50	2.85	2.15	1.8	1.65	1.2	0.5

Table 7: The height information. Notice that when the heights of several different obstacles are almost the same, we adopt the average value for clearness.

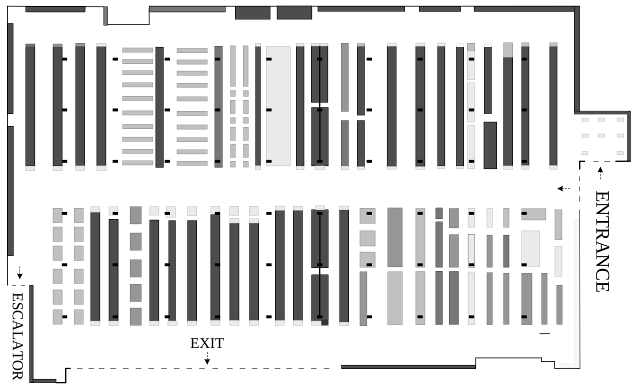


Figure 11: The plane layout of the shopping area. The geometries filled in different colors are with different heights.

that of other two ones, when the number of edges increases. Compared to the 'Standard' method, the better performance of our proposed method is ascribed to those optimization strategies, and the poorer performance of the 'Naive' method is due to that computing the intersections in such a way is (somewhat) inefficient. Especially, this deficiency is more obvious when the number of edges of polygons is large.

10.4. Case Study

In this subsection, we study our proposed algorithm based on a specific application. We use the 'Auchan supermarket' located at 2092 Dongchuan Rd., Shanghai, China, as a sample. We collect the *deployment information* of the shopping area on the 2nd floor of this supermarket. Figure 10(a) exposes a corner of this shopping area. The length and width of the shopping area are 100 and 60 meters, respectively. Its plane layout is depicted in Figure 11 for the sake of intuition. The heights of different shelves or obstacles are listed in Table 7.

To connect our algorithm with the specific application, we consider a set of n_c free-rotating cameras which are to be placed on the ceiling for monitoring the shopping area; later, we shall employ our algorithm to check the blind angles. In the following, we restrict our attention to checking the blind angles on the ground, i.e., the '$z = 0$' plane. (Remark: the blind angles for the '$z \neq 0$' plane can be achieved similarly.)

In our experiments, we adopt two types of manners to deploy the n_c cameras: (i) they are placed uniformly; and (ii) they are placed randomly. For short, we denote by **U** and **R** these two distributions, respectively. We regard the visual range of a single camera as a circle with the center c and radius r when there is no obstacle near to it, where the x- and y- coordinates of c are the same as the ones of the camera but $z = 0$. Otherwise, we compute its visual range by considering the effect of obstacles. See Figure 10(b) for an illustration of computing the visual range.

Let B_a, S_a, O_a, and V_a^i denote the blind angles (notice: here we do not consider the areas occupied by obstacles), the shopping area, the areas occupied by obstacles such as shelves, and the ith camera's visual range respectively, where $i \in [1, \cdots, n_c]$. We compute B_a based on the following equation.

$$B_a = (S_a - O_a) - \bigcup_{i=1}^{n_c} V_a^i$$

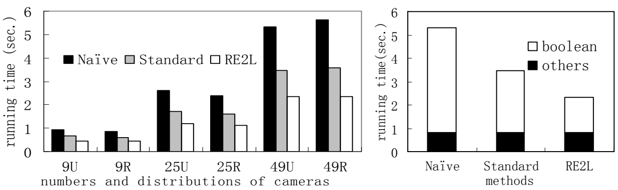
For ease of reference, Table 8 lists the parameters used in our experiments. We use the 'Standard' and the 'Naive' methods as the baselines when we compute the blind angles, as both of them work well for the floating point data type. Figure 12(a) reports the performance results when we use different settings. We can easily see that the execution time increases for all the methods when we vary the number of cameras from 9 to 49. This is mainly because for those 'new' cameras we also need to compute their visual areas and to remove them by boolean operations, and so the total number of boolean operations needed to be executed increases, which consumes more computation time. From these figure, we realize another interesting phenomenon, i.e., the distributions of cameras make little impact on the execution time (remark: here the distributions refer to *uniform* and *random*, instead of others). Besides the above two findings, we can also see that our preferred method, RE2L, always outperforms its competitors for all these settings. This further validates the superiority of our algorithms. Moreover, Figure 12(b) reports the time consumed by boolean operations and the one consumed by all other operations. This shows us that improving the efficiency of boolean operations is feasible and also important for this type of applications.

11. Conclusions

In this paper we investigated the problem of boolean operation on circular-arc polygons. By well considering the nature

Para.	Desc.	Value
n_c	number of cameras	9, 25, 49
r	radius of the visual range	5
h	height of cameras being placed	4
D	distribution of cameras	U, R

Table 8: The parameters used in this set of experiments.



(a) running time vs. n_c and D (b) $n_c = 49$ and D =U

Figure 12: Pefromances for computing the blind angles.

of the problem, concise data structure and targeted algorithms were proposed. The proposed method runs in time $O(m+n+(l+k)\log l)$, using $O(m+n+k)$ space. We conducted extensive experiments demonstrating the effectiveness and efficiency of the proposed method, and showed our techniques can be easily extended to compute boolean operation of other types of polygons. We conclude this paper with two research topics: (i) As we know, the multiprocessor and multi-GPU systems are widely used nowadays; it could be interesting to design efficient parallel algorithms for computing boolean operations of polygons. (ii) As we showed in the paper, our techniques can be easily extended to compute boolean operations of traditional polygons, and circular-ac polygons with holes and self-intersections; it is still open whether our techniques can be extended to compute boolean operations on conic polygons or more general cases.

Acknowledgments

This work was supported by the National Basic Research 973 Program of China (No. 2015CB352403), the NSFC (No. 61272032, 61202024, 61202025, 61502220 and U1304616), the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939), Shanghai Pujiang Program (No. 13PJ1403900), Singapore NRF (CREATE E2S2), the Program for Changjiang Scholars and Innovative Research Team in University of China (IRT1158, PCSIRT).

References

[1] R.D. Andreev, Algorithm for clipping arbitrary polygons, Computer Graphics Forum (CGF) 8 (1989) 183–191.

[2] J.L. Bentley, T. Ottmann, Algorithms for reporting and counting geometric intersections, IEEE Transactions on Computers (TC) 28 (1979) 643–647.

[3] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, E. Schmer, A computational basis for conic arcs and boolean operations on conic polygons, in: European Symposium on Algorithm (ESA), 2002, pp. 174–186.

[4] E. Berberich, M. Hemmer, M. Kerber, A generic algebraic kernel for nonlinear geometric applications, in: Symposium on Computational Geometry (SoCG), 2011, pp. 179–186.

[5] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational geometry: algorithms and applications, Third Edition, Springer, Berlin, 2008.

[6] I.Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, E.P. Tsigaridas, Towards and open curved kernel, in: Symposium on Computational Geometry (SoCG), 2004, pp. 438–446.

[7] E. Fogel, D. Halperin, R. Wein, CGAL Arrangements and Their Applications: A Step-by-Step Guide. Geometry and Computing, Springer, 2012.

[8] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics: principles and practice, Second Edition, Addison-Wesley, Massachusetts, 1996.

[9] Y. Gardan, E. Perrin, An algorithm reducing 3d boolean operations to a 2d problem: concepts and results, Computer-Aided Design (CAD) 28 (1996) 277–287.

[10] Y.X. Gong, Y. Liu, L. Wu, Y.B. Xie, Boolean operations on conic polygons, Journal of Computer Science and Technology (JCST) 24 (2009) 568–577.

[11] G. Greiner, K. Hormann, Efficient clipping of arbitrary polygons, ACM Transactions on Graphics (TOG) 17 (1998) 71–83.

[12] C.M. Hoffmann, J.E. Hopcroft, M.S. Karasick, Robust set operations on polyhedral solids, IEEE Computer Graphics and Applications 9 (1989) 50–59.

[13] D. Krishnan, L.M. Patnaik, Systolic architecture for boolean operations on polygons and polyhedra, Computer Graphics Forum (CGF) 6 (1987) 203–210.

[14] Y.D. Liang, B.A. Barsky, An analysis and algorithm for polygon clipping, Communications of the ACM (CACM) 26 (1983) 868–877.

[15] Y.K. Liu, X.Q. Wang, S.Z. Bao, M. Gombosi, B. Zalik, An algorithm for polygon clipping, and for determining polygon intersections and unions, Computers and Geosciences (GANDC) 33 (2007) 589–598.

[16] P.G. Maillot, A new, fast method for 2d polygon clipping: Analysis and software implementation, ACM Transactions on Graphics (TOG) 11 (1992) 276–290.

[17] A. Margalit, G.D. Knott, An algorithm for computing the union, intersection or difference of two polygons, Computers and Graphics (CG) 13 (1989) 167–183.

[18] F. Martinez, A.J. Rueda, F.R. Feito, A new algorithm for computing boolean operations on polygons, Computers and Geosciences (GANDC) 35 (2009) 1177–1185.

[19] S. Meguerdichian, F. Koushanfar, M. Potkonjak, M.B. Srivastava, Coverage problems in wireless ad-hoc sensor networks, in: IEEE International Conference on Computer Communications (INFOCOM), 2001, pp. 1380–1387.

[20] Y. Peng, J.H. Yong, W.M. Dong, H. Zhang, J.G. Sun, A new algorithm for boolean operations on general polygons, Computers and Graphics (CG) 29 (2005) 57–70.

[21] A. Rappoport, An efficient algorithm for line and polygon clipping, The Visual Computer (VC) 7 (1991) 19–28.

[22] M. Rivero, F.R. Feito, Boolean operations on general planar polygons, Computers and Graphics (CG) 24 (2000) 881–896.

[23] M.I. Shamos, D. Hoey, Geometric intersection problems, in: IEEE Symposium on Foundations of Computer Science (FOCS), 1976, pp. 208–215.

[24] I.E. Sutherland, G.W. Hodgman, Reentrant polygon clipping, Communications of the ACM (CACM) 17 (1974) 32–42.

[25] B.R. Vatti, A generic solution to polygon clipping, Communications of the ACM (CACM) 35 (1992) 56–63.

[26] B. Wang, Coverage problems in sensor networks: A surveys, ACM Computing Surveys (CSUR) 43 (2011) 1–53.

[27] Z.J. Wang, D.H. Wang, B. Yao, M. Guo, Probabilistic range query over uncertain moving objects in constrained two-dimensional space, IEEE Transactions on Knowledge and Data Engineering (TKDE) 27 (2015) 866–879.

[28] Z.J. Wang, B. Yao, R. Cheng, X. Gao, L. Zou, H. Guan, M. Guo, Sme: explicit & implicit constrained-space probabilistic threshold range queries for moving objects, Geoinformatica 20 (2016) 19–58.

[29] K. Weiler, P. Atherton, Hidden surface removal using polygon area sorting, in: International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), 1977, pp. 214–222.

[30] R. Wein, Exact and approximate construction of offset polygons, Computer-Aided Design (CAD) 39 (2007) 518–527.