Noname manuscript No.
(will be inserted by the editor)

Windowing Queries Using Minkowski Sum and Their Extension to MapReduce

Sepideh Aghamolaei · Vahideh Keikha · Mohammad Ghodsi · Ali Mohades

Received: date / Accepted: date

Abstract Given a set of n segments and a query shape Q, the windowing length query asks for finding the length of the segments that lie inside Q. For square queries, a $O(n^2)$ time algorithm and a matching lower bound exist. We solve this problem on convex polygons and disks as query shapes, with $O(\log n)$ query time and polynomial preprocessing time. Using our data structure, we solve the problem of finding popular places in a set of trajectories.

Other than reporting queries, we use computing the sum of lengths of segments inside the query shape, called the length query, and define a variation of the problem of finding the popular places based on the length of the trajectories inside the query shape.

We also give algorithms for computing the length query for c-packed curves, and use it to approximate the minimum value c for which a curve is c-packed, if such a c exists.

S. Aghamolaei
Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.
Tel.: +9899 1251 5184
Fax: +9821 6616 5048
E-mail: aghamolaei@ce.sharif.edu

V. Keikha
Laboratory of Algorithms and Computational Geometry, Department of Mathematics and Computer Science, Amirkabir University of Technology.
Department of Computer Sciences, University of Sistan and Baluchestan, Zahedan, Iran.
E-mail: va.keikha@aut.ac.ir

M. Ghodsi
Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.
School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran. E-mail: ghodsi@sharif.edu

A. Mohades
Laboratory of Algorithms and Computational Geometry, Department of Mathematics and Computer Science, Amirkabir University of Technology. E-mail: mohades@aut.ac.ir

Our results extend to MRC and MPC models for MapReduce, where we address these problems on a set of x-monotone curves. The round complexities of our MapReduce algorithms are constant.

In addition, we also implemented our popular places algorithms on trajectories on inputs as big as 15K points to evaluate the efficiency of our algorithms in practice.

Keywords Windowing Queries · Trajectories · Finding Popular Places · c-Packed Curves · Approximation Algorithms · MapReduce

1 Introduction

In this paper, we introduce a new notion of querying on a set of line segments called the "length query problem": Suppose we are given a set of line segments, which we call S, and suppose we are also given a query object Q. The goal is to compute the total lengths of parts of the segments in S that are contained in Q.

A closely related class of well-known problems is the *windowing range queries*, in which S is a set of n segments, and the aim is to preprocess the points into a data structure so that for a query range Q, the segments intersecting Q can be *reported* or *counted* efficiently. The length query can be computed in $O(k)$ time, on the output of the reporting version, if the number of intersected segments is k.

The rectangular windowing query can be answered in $O(\log^2 n+k)$ time, on n interior-disjoint segments and a rectangular query using a segment tree and a binary search tree [9]. The preprocessing time is $O(n \log n)$ and the total space is also $O(n \log n)$. For arbitrary segments, one has to eliminate their intersections at points other than their endpoints before building the data structure. This can be done by computing the arrangement of those segments. We improve the query time to $O(\log n + k)$ for arbitrary curves, while keeping the preprocessing time polynomial in n and k (See Section 2).

Our idea is to use Minkowski sum, a concept mostly known for its use in motion planning, to solve the windowing queries. Assume a fixed point of the shape is chosen as the *representative point*, which can be used to uniquely identify a translation of the shape. By summing the set of input shapes with the query shape, we get the regions containing the representative points of the shapes that intersect with the input shapes. While Minkowski sum has been used for computing the arrangement of shapes [12], to the best of our knowledge, it has not been used in range queries before. Arrangement of congruent disks and their applications has also been studied [8]. We refer the reader to [32] for more information about constructing the arrangements of shapes.

Our methods introduce a theoretical framework for a new class of query-based problems, where the aim is to preprocess the input into a data structure based on the arrangement of shapes to compute a function of a set of objects intersected by a query shape, also known as a window, using point location queries. Polygonal point location queries can be answered in $O(\log n)$ time on an arrangement of n line segments [32]. Experimental methods exist for point location in the arrangement of arbitrary arcs [19, 13]. We solve the point location queries on the arrangement of a set of n congruent disks in $O(\log n)$ time, with $O(n^3 \log n)$ preprocessing time.

A family of curves, called c-packed curves [10], have the property that for any disk, the length of the curve inside the disk is at most c times the radius of the disk. We also give an algorithm for approximating the minimum value c for which a polygonal curve is c-packed and an $O(n \log n)$ approximation algorithm for computing length queries on such curves.

For a given set of points, we define *r-restricted near neighbors* as the problem that asks for the set of points within distance r of a query point q. *Locality-sensitive hashing* is a method for solving near neighbors problem in high dimensions [30], where a hash function is used that takes r as input and finds all points within distance cr. Then, the approximate nearest neighbor problem can be solved by hashing the query point and checking all the points with the same hash value. Our algorithm for point location in a set of congruent disks solves the r-restricted near neighbors problem.

Barequet et al. [3] introduced a problem in which the input S is a set of n points, and the aim is to find a translation of Q that maximizes the number of points contained by the translation of Q. For a convex polygon with m vertices, the problem can be solved in $O(nk \log(mf) + m)$ time and linear space, where f is the maximum number of points in a translation of Q, i.e. the output size [3]. In the case where the convex polygon is an axis-parallel rectangle, finding an optimal translation to cover the maximum sized subset takes $O(n \log n)$ time [27]. By computing the Minkowski sum of Q with the input points, the problem converts to an instance of counting intersections problem discussed in the current paper.

Finding popular places was introduced by Benkert et al. in 2007 [5], inspired by applications in a previously known empirical class of problems called the *convergence patterns* [23, 24]. In particular, they defined two variations of the problem, namely the *discrete* and the *continuous* popular places problem. The discrete version of the problem asks for finding squares of a fixed size such that at least f vertices, each from a distinct trajectory, lie inside those squares. In the continuous popular places problem, they look for a square that intersects with at least f distinct trajectories. With n as the total complexity of the trajectories, they proposed a line-sweeping algorithm with $O(n \log n)$ time complexity for the discrete version of this problem, and quadratic time complexity for the continuous version. In both cases, the space complexity is linear in n.

Fort et al. [14] experimentally studied the problem of the popular places on trajectories for the disks of a given radius, and a given value of f as a threshold of the popularity. Their idea was sweeping the edges of the trajectory by a disk of radius r that is centered at the trajectory, and computing the intersections between the result of the sweeping that they call *popularity map*. Then they round the popularity map to a grid with fix resolution. Also, they have shown that since the problem can be modeled on a grid, if they use a CUDA $2D$

grid for the problem, they can achieve a good parallelism for the problem of popular places on a Graphics Processing Unit (GPU). However, the grid idea cannot be applied to all trajectories since it is based on the assumption that knowing whether each cell of the grid is a popular place or not is enough, which imposes a rounding error on the reported solution. On the other hand, CUDA grids have a bounded size. Furthermore, the authors did not analyze the time complexity of any of the algorithms, which is critical in determining the applicability of the algorithm to huge datasets.

Our findings imply that on a data-set of dense trajectories, the grid-based method [14] can have a good solution in practice. It is because in a dense set of trajectories many cells are popular, and each point that belongs to a popular region lies in such a cell with a good probability. If the data-set is not dense, with a query of any arbitrary shape, our algorithms give the exact solutions in a reasonable time.

Definition 1 (Finding Popular Places [5]) A set S of n line segments, an integer f, and a query shape Q are given. The goal is to find a translation of Q so that at least f segments from different trajectories are intersected by Q.

Using Minkowski sum of the square and the input trajectories, this problem converts to an instance of counting intersections problem.

With the same application in mind as the popular places problem, the popularity of a place can be seen as the amount of time spent in that area, which can be formulated as the length of the curve inside a specific region. Note that one cannot simply adjust the sweeping technique of [5] to this problem since unlike the popular places problem, discretizing the problem on the vertices does not give the events.

In case of practical applications, as the aim is to retrieve relevant data from a given data set, the length query problem has a wide range of applications, e.g., in animals flock patterns to determine the duration in which animals stay in a specific region, where they seek out places that have warmth, food and are safe for breeding; see, e.g., [33] and the references therein.

Contributions

Inspired by the popular places problem, we define the *maximum-length popular place problem* where the input is a set of line segments and the aim is to preprocess the line segments such that the total lengths of the segments inside the query can be computed faster than checking every line segment to see if it intersects with the query. More formally, we study the following problems:

Definition 2 (Length Query Problem) A set $S = \{s_1, \ldots, s_n\}$ of n line segments and a query shape Q of complexity m are given. The goal is to compute the sum of lengths of parts of the segments of S that are inside Q.

Figure 1 illustrates a length query where the shape of the query is a disk.

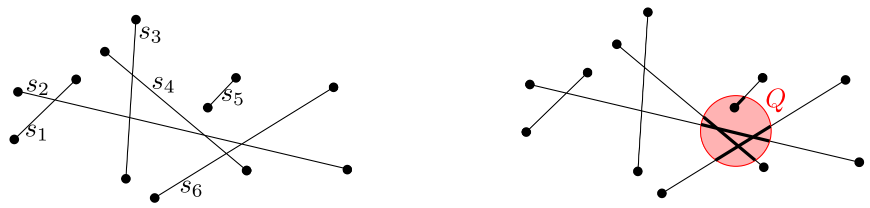

Fig. 1 The input is $S = \{s_1, \ldots, s_6\}$, and the query Q (here, a disk) is shown in red. The total sum of the lengths of thick segments determines the solution to the length query problem.

Definition 3 (Maximum-Length Popular Place) A set P of K trajectories of total complexity n and a query shape Q are given. The goal is to find a translation of Q so that at least f distinct trajectories are intersected by Q, and for each of the intersecting trajectories, the length of the part of the intersecting segment inside Q is at least t.

We also discuss connections among the length query problem and some of the existing problems and introduce the query versions of those problems. In this paper, we achieve the following results:

– Our method can speed up the query time of the windowing problem. We first compute the Minkowski sum of the input and the rectangular query, then we build the arrangement of the resulting shapes, and assign to each cell, the set of segments intersecting the boundary of that cell. Then, the windowing problem reduces to a point location problem, and with $O(n^3 \log n)$ preprocessing time and $O(n^2)$ space, we are able to answer the query in $O(\log n + k)$ time. Note that we can apply this method to any convex polygonal query of complexity m, achieving query time $O(\log n + \log m + k)$, with $O(n^3 m^2 \log(mn))$ preprocessing time and $O(n^2 m^2)$ space.
– Similar as above, in the popular places problem, if we compute the Minkowski sum of the trajectories and the shape Q of complexity m, and then during the construction of the arrangement we keep a cell c with the maximum number of distinct vertices and intersected segments. The popular places problem in both the continuous and the discrete models is then reduced to finding such a cell c. Such a point location query can be answered in $O(n^3 m^2 \log(mn))$ time for problems with convex polygons as query shapes (Section 2.4). It is a significant improvement since to date, to the popular places problem which has not been studied for shapes other than a square of fixed size.
– We design an exact algorithm that preprocesses a set of n segments into a data structure with $O(n^3 m^2)$ space and $O(n^3 m^2 \log(mn))$ preprocessing time, such that the exact solution to the length query problem can be computed in $O(\log n + \log m + k)$ time, where k is the number of intersecting segments (Section 2).

- We design an approximation algorithm for the maximum-length popular places problem that runs in $O(m^2n^3\log(mn)+m^3n^2)$ time (Section 2.2).
- For a polygonal curve of complexity n, we give an approximation algorithm for the minimum c for which the curve is c-packed, if such a constant c exists (Section 4.1).
- For c-packed curves, we give a $2c$-approximation algorithm for the length query problem in Section 4.
- We give a MapReduce algorithm based on our sequential algorithm for popular places, which takes $O(\log_m n)$ rounds and $O(nk)$ space, on k x-monotone polygonal curves (Section 5).
- We also evaluate the efficiency of our algorithms in practice by implementing the sequential and the MapReduce algorithms of the popular places problem and analyzing the results on a big data-set (Section 6).

A summary of the theoretic results is presented in Table 1.

Windowing Problem	Query Shape	Preprocess	Space	Query Time	App	Refs
interior-disjoint segments	axis-parallel rectangle	$O(n\log n)$	$O(n\log n)$	$O(\log^2 n+k)$	1	segment tree [9]
(reporting)	convex polygon	$\tilde{O}(n^3m^2)$	$O(n^2m^2)$	$O(\log(nm)+k)$	1	Lemmas 2 and 3
(counting)	"	$\tilde{O}(n^3m^2)$	$O(n^2m^2)$	$O(\log(nm))$	1	Lemmas 2 and 3
-	"	$\tilde{O}(n^3m^2+n^2m^3)$	$O(n^2m^2)$	$O(\log(nm)+k)$	1	Theorems 3 and 4
Popular Places	squares	$\tilde{O}(n^2)$	$O(n^2)$	-	1	[5]
Popular Places	convex polygon	$\tilde{O}(n^3m^2)$	$O(n^2m^2)$	-	1	Theorem 5
Max-Len Popular Places	convex polygon	$\tilde{O}(n^3m^2+n^2m^3)$	$O(n^2m^2)$	$O(\log(nm)+k)$	1	Theorem 3
c-packed curves	disk	$O(n\log n)$	$O(n)$	$O(\log n)$	$2c$	Section 4

Table 1 A summary of the results for length query and related problems. L is the length of the input curve dividing by the length of the shortest edge of the curve, k is the number of intersecting segments of the input curve with the query shape, m is the complexity of the query shape, and n is the number of vertices in the input curves. $\tilde{O}$ hides polylog factors.

The summary of results in MapReduce is shown in Table 2. These results are under MRC model and assume the number of intersections is at most $O(nk)$, if any vertical line intersects with at most k segments.

Preliminaries

In the following, we review several necessary definitions.

Windowing Problem	Type	Query Shape	Rounds	Space	Reference
K x-monotone curves	reporting	convex polygon	$O(\log_M n)$	$O(nK)$	Theorem 12
“	popular places	convex polygon	$O(\log_M (nm))$	$O(nmK)$	Theorem 13

Table 2 A summary of the results for length query and related problems in MRC. Here, m is the complexity of the query shape, M is the memory of each machine, ℓ is the number of machines, and n is the number of vertices in the input curve.

Minkowski Sum Let A and B be two polygonal regions in the plane. The *Minkowski sum* of them is defined as [6]

$$A \oplus B = \{a + b | a \in A, b \in B\}$$

in which $a + b$ is just a vector sum. The Minkowski is the most commonly used tool in the analysis of translational motion planning; it is frequently convenient to compute the Minkowski sum of the robot and the obstacles in a workspace, and then consider the robot as a single point (See, e.g., [22, 31]). We also use a similar interpretation in our analysis.

Suppose A and B has m and n vertices, respectively. The combinatorial explosion of $A \oplus B$ depends on both A and B, i.e., if both A and B are convex, then $A \oplus B$ has $m + n$ edges, but if one is convex and the other is a simple polygon, the complexity of $A \oplus B$ is $O(mn)$, and in the case where both A and B are simple, the complexity grows to $O(m^2n^2)$ [21, 28] (See also [26]). However, in the general case, where we have no assumptions on A and B, the complexity of a single face of $A \oplus B$ can be $\Theta(mn\alpha(min\{m, n\}))$, where $\alpha(.)$ is the functional inverse of Ackermann's function.

Point Location Given is a partition of the plane into n *polygonal faces* with disjoint interiors, the *point location* problem asks for the face containing a query point that is specified by its coordinates. In the case where the faces are convex [29], or concave but monotone [11], in a specific direction, the solution to the point location problem can be found in $O(\log n)$ time, after spending $O(n \log n)$ preprocessing time, and using $O(n)$ space. Since preprocessing takes $O(n)$ time on a monotone subdivision; an arbitrary planar subdivision can be made monotone by a plane sweep in $O(n \log n)$ time, and then the algorithm of [29] can be adjusted to work with arbitrary subdivisions. We refer the reader to [2] for further references.

c-Packed Curves The notion of *c-packed* curves was introduced by Driemel et al. [10] in 2010 with the aim of computing a $(1+\epsilon)$-approximation algorithm for the Fréchet distance in $\mathbb{R}^d$ between two polygonal curves, where the algorithm runs in $O(cn/\epsilon + cn \log n)$ time, and a curve is c-packed if the total length of the curve inside any ball of radius r is at most cr. Due to the realistic input assumptions of c-packedness, several researchers used this notion to design efficient algorithms for the problems with curves as input, see, e.g., [7].

Arrangement For a given set X of geometric objects in the plane, the intersections between the elements of X make a subdivision consisting of vertices, edges and faces. Let $\mathcal{A}(X)$ denote such subdivision that is called the *arrangement* of X. It is already studied that for a set Z of n pseudo-lines, i.e. two curves that can intersect at most once, the maximum number of the vertices, edges or faces of $\mathcal{A}(X)$ is bounded by $O(n^2)$. See [32] for more details.

The arrangement of a set of Jordan arcs in the plane can be computed by an output-sensitive algorithm with running time $O((n+k)\log n)$ time and $O(n+k)$ space, in which k denote the number of intersection points in the arrangement [32]. The idea is a sweep-line technique that incrementally produces the output. The arrangement of a set of Jordan arcs can also be computed in parallel, taking $O(\log n)$ time and $O(n^2)$ processors [17]. There also exists an output-sensitive parallel algorithm for computing the arrangement that runs in $O(\log^2 n)$ time with taking $O(n + \frac{k}{\log} n)$ processors [16].

Core-sets and ϵ-kernel A core-set for a function f on a point-set Z is a set $C \subset Z$, such that $f(C)$ approximates $f(Z)$. ϵ-kernel [1] is a core-set for the extent measure of a point-set, where given a set of points, the farthest point in a set of equidistant directions is computed. They require $O(\frac{1}{\sqrt{\epsilon}})$ directions to compute a $(1+\epsilon)$-approximation of the distance.

MapReduce MapReduce is a parallel and distributed computing framework, in which a set of independent machines work in parallel rounds and can communicate after each round. Theoretical models for MapReduce have been proposed [20, 4] to define the efficiency of a MapReduce algorithm. In the MRC (MapReduce Class) model [20], the number of machines ℓ and the memory of each machine M should both be sublinear in the input size, and the number of rounds should be polylogarithmic. In the MPC (Massively Parallel Computation) model [4] the number of machines ℓ and the memory of each machine M is restricted to be sublinear in the input size, and the total memory to be linear, and the number of rounds to be constant. We use both models in our paper, depending on the output size of the problem. If the output does not fit inside the total available memory $M\ell$, the problem cannot be solved in these models. Sorting and parallel prefix computations take $O(\log_M n)$ rounds in MapReduce [18]. CRCW PRAM can be simulated in MapReduce with slowdown 2 for the round complexity [18]. Data distribution can be solved using parallel prefix [25]. Current frameworks for MapReduce implement partitioning via hashing, which achieves the same bound but is randomized.

2 An Exact Algorithm for The Length Query Problem

In this section, we show that we can preprocess a set of n segments S into a data structure with $O(n^3m^2)$ space and $O(n^3m^2\log(mn))$ preprocessing time, such that for any convex polygonal query of complexity m, the solution to the length query problem can be computed in $O(\log n + \log m + k)$ time, where k is the number of intersecting segments.

2.1 Intersection Query: The Minkowski Sums of The Segments with The Query Shape

By computing the Minkowski sum of the query shape Q and each segment in the input, we get a set of polygons. If the representative point q of the query shape Q falls inside one of these polygons, it means Q intersected the segment that corresponds to that polygon. To precompute the set of segments intersecting the same query shape, we define a partitioning of the plane, which we call an **AQD**:

Definition 4 (Aggregated Query Diagram of a set of segments (AQD)) For a set S of n line segments and a query shape Q, we call a partitioning $c_1, \ldots, c_k$ of the plane with a set $O_i \subset S$ at each c_i, an aggregated query diagram if for any point $q \in c_i$, the query shape with representative point q intersects the subset O_i of segments, and k is minimized.

Note that each AQD is for a unique query shape, and it can only be translated (See Figure 2). A fixed point q of the query shape $Q(q)$ is used as the representative point for each translated copy. This point can be used to map the solution from the AQD to the Euclidean plane.

2.1.1 The Mapping Between AQD and The Euclidean Plane

We restrict the Minkowski sum to use the representative point and not all points of the query shape. The inverse of such a sum that maps a point p in the AQD to a point in the Euclidean plane is the set of representative points q of all query shapes $Q(q)$ containing p. To make such a sum uniquely reversible, choose the leftmost point of the shape, or for symmetric shapes their center as the representative point.

In the first case, a point p in AQD can be mapped to a point q in the Euclidean plane by placing the rightmost point of the shape Q on p and report the leftmost point of Q as q. Since the query shapes are convex, the leftmost/rightmost points are either a vertical line segment or a single point, both of which can be mapped back uniquely by mapping back the endpoints of the segment or the single point.

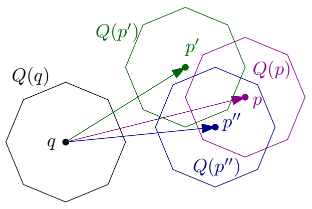

Fig. 2 A query Q and several of its translations.

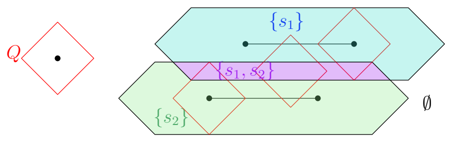

Fig. 3 An AQD for two segments s_1, s_2 and a square query shape. The AQD has four different cells, where one of them -the outer face- corresponds to the empty set.

The second case, where the center of a symmetric shape with respect to that center is chosen as its representative point, the AQD maps to itself. An example of an AQD for a unit square is shown in Figure 3.

Algorithm 1 builds an AQD using an algorithm for computing the arrangement of a set of segments, which is then used in Algorithm 4 as a planar subdivision in which we do point-location. Algorithms with $O(n \log n)$ pre-

Algorithm 1 Building An AQD

```
Input: A set of segments $S$, a convex polygon $Q$
Output: A data structure for point location
1: for each segment $s_i \in S$ do
2:     $P_i$ = compute the Minkowski sum of $Q$ and $s_i$
3: $\{c_j\}_{j=1}^{k}$ = compute the arrangement of $P_i, i = 1, \dots, n$
4: using a line sweeping algorithm, compute and store sets $O_i$, for $i = 1, \dots, k$ in each cell
   $c_i$.
5: build a point-location data-structure on $\{c_j\}_{j=1}^{k}$.
```

processing time, $O(\log n)$ query time, and $O(n)$ space for point location in a planar subdivision exist (See [32]).

Theorem 1 *Algorithm 1 computes an AQD of S.*

Proof Any point q that lies inside the polygon P_i is the representative point of a query shape Q that intersects with s_i, based on the properties of Minkowski sum. The opposite also holds: for each segment s_i that intersects with a query Q' with representative point q', its corresponding polygon P_i contains q'. So, by constructing a diagram for points that are inside the same set of polygons P_i, for $i = 1, \dots, n$, we also get an AQD for S. Such a diagram is an arrangement of polygons P_i, for $i = 1, \dots, n$.

Note that different cells of an AQD can have the same set of intersecting segments, i.e. it is possible that for $i, j \in \{1, \dots, k\}, i \neq j : O_i = O_j$.

In Lemma 1, we give a bound on the complexity of AQD.

Lemma 1 *The number of cells in an AQD created by a query shape of complexity m is $O(n^2m^2)$, and its total size is $O(n^3m^2)$.*

Proof The total number of segments in the arrangement is nm, since there are n polygons, each with m vertices. So, they intersect in at most $\binom{nm}{2}$ points, and their arrangement has a size at most $\binom{nm}{2} = O(n^2m^2)$. Each cell of the diagram stores the set of intersecting segments from S, so the total size is

$$\sum_{i=1}^{k} |O_i| \leq k \max_{i=1,\dots,k} |O_i| \leq kn \leq \binom{nm}{2} n = O(n^3m^2).$$

Remark. The output-sensitive complexity of AQD is $O(n^2m \log(nm) + nk)$ based on the output-sensitive complexity of the line-arrangements [32] which can be less than the bound of Lemma 1. CREW PRAM algorithms with polylogarithmic complexity also exist, even for the output-sensitive case [17, 16], which can be simulated in MapReduce with slowdown 2.

Lemma 2 *The time complexity of Algorithm 1 is* $O(n^3m^2\log(mn))$*.*

Proof Computing the Minkowski sum of the segments and the query shape can be done in $O(nm)$ time. Building the arrangement of mn segments takes $O(n^2m^2\log(mn))$ time. Sweeping the arrangement and assigning sets O_i to their corresponding cell c_i takes $O(n^3m^2\log(m^2n^2))$ time, since the size of each element in the queue can be $O(n)$, so each update takes $O(n\log(n^2m^2))$ time. The time complexity of the sweeping step dominates the other time complexities of the algorithm, so, the overall time complexity is $O(n^3m^2\log(m^2n^2))$.

Algorithm 2 Reporting Query

```
Input: AQD = The output of Algorithm 1, a query $Q$
Output: The set of segments that are intersecting with $Q$
1: $c_j$ = Do a point location on AQD for the representative point of $Q$.
2: return $O_j$
```

Lemma 3 *Algorithm 2 solves the counting problem in* $O(\log n + \log m)$ *and the reporting problem in* $O(\log n + \log m + k)$ *time, where* k *is the output size.*

Proof Algorithm 2 does a point location on a polygonal subdivision of size $O(m^2n^2)$, so it takes $O(\log(m^2n^2)) = O(\log m + \log n)$ time for finding the cell c_j which has a pointer to the solution set O_j. Reporting the set of intersecting segments (O_j) takes $|O_j|$ time, which is the size of the output.

2.2 Precomputing The Sum of The Lengths

Computing the exact length of the segments intersecting a given query can be done by summing up the lengths of the segments inside the query shape. Using AQD and point-location, only the intersecting segments need to be checked. However, finding the points with query length at least f requires pre-computing the lengths and bounding them.

In this section, we give an algorithm for computing the formula for the length of the curve inside each cell, in terms of the representative point q of the query $Q(q)$ and the edges of the input curve.

The length of a query after taking the Minkowski sum remains the same, since the query shape and all the input curves have been translated in the same direction by the same amount. Therefore, it is enough to compute the length queries on the AQD. Mapping back the vertices of AQD, as explained in Section 2.1.1, is also possible.

Before computing the maximum length inside the query, we need to determine the position of the query shape with respect to each segment. There are four possible placements that can affect the formula for length computation, assuming without loss of generality, that the segment is horizontal:

- d_1: the query shape intersects with the segment, but none of the endpoints are inside the query.
- d_2: only the right endpoint lies inside the query shape.
- d_3: the segment lies completely inside the query shape.
- d_4: only the left endpoint lies inside the query shape.

Note that only one of d_1 and $d3$ can happen at the same time for a segment, depending on whether the length of the segment is less or more than the length of the shape. An example of this finer partitioning of an AQD cell is given in Figures 4 and 5.

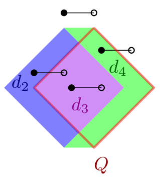

Fig. 4 The case with $d_1 = \emptyset$ in Algorithm 3.

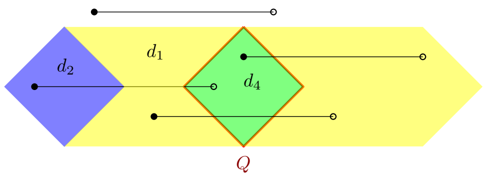

Fig. 5 The case with $d_3 = \emptyset$ in Algorithm 3.

Lines 8-15 of Algorithm 3 compute the length of the segment based on these placements. Let $\phi(c_j)$ be the finer partitioning into sub-divisions of cell c_j with regions corresponding to each of the 4 above placements (Line 16 of Algorithm 3).

Let i be the perpendicular direction from point p to line s_i and j be perpendicular to i. The length of s_i inside Q starts from 0 where the closest point of the shape touches s_i and ends again with 0 when the farthest point of the shape touches s_i. At any point in between, the length of s_i inside Q can be parameterized by its distance from a fixed point, for which we use the farthest endpoint of s_i in perpendicular direction.

Theorem 2 *Algorithm 3 solves the* finding length-f popular places *problem.*

Proof Assume $Q(O)$ is a translation of Q whose representative point lies on the origin. Each point on the boundary of a translation of Q can be defined by the following formula:

$$Q(q) = q + Q(O).$$

Functions $F_{j,d}(.)$ are the sum of a set of Euclidean distances between the points. In Line 19 of Algorithm 3, we compute the solutions of this sum, which are of the following form:

$$\cup_{j=1}^{k}\{q| \sum_{s_i \in O_j} \sqrt{(x(q)-\alpha_i)^2 + (y(q)-\beta_i)^2} \geq f\},$$

Algorithm 3 Preprocessing Sum of Lengths

```
Input: A set of input curves $P$, a constant $f > 0$
Output: The geometric locus of the representative of the popular places
 1: $C$ = the AQD of the segments of the curves in $P$.
 2: for each cell $c' \in C$ do
 3:     for $s_i \in O_j$ do
 4:         $Q'$ = the rotation of $Q$ with the same clock-wise rotation that makes $s_i$ parallel
    to the $x$-axis.
 5:         $Q_R$ = the point of $Q'$ with maximum $x$-value at each $y$-value.
 6:         $Q_L$ = the point of $Q'$ with minimum $x$-value at each $y$-value.
 7:         $p_i = (x_i, y_i)$ : the left endpoint of $s_i$, and $p'_i = (x'_i, y'_i)$ the right endpoint of $s_i$.
 8:         if $p_i, p'_i \notin Q'$ then
 9:             $\mathrm{dist}_{i,1} = d(Q_R(y_i), Q_L(y_i))$
10:         else if $p_i \notin Q', p'_i \in Q'$ then
11:             $\mathrm{dist}_{i,2} = d(Q_L(y_i), p'_i)$
12:         else if $p_i, p'_i \in Q'$ then
13:             $\mathrm{dist}_{i,3} = d(p_i, p'_i)$
14:         else if $p_i \in Q', p'_i \notin Q'$ then
15:             $\mathrm{dist}_{i,4} = d(Q_R(y_i), p_i)$
16:         $\phi(c_j)$ = divide $c_j$ into regions each corresponding to one of the 4 if-statements
    of lines 8-15.
17:         $f_{i,d}(.)$ = sum of $\mathrm{dist}_{i,d}$ for $(p_i, p'_i) \in O_j, d \in \phi(c_j)$ on $q \in c_j$.
18:     $F_{j,d}(.) = \sum_{s_i \in O_j} f_{i,d}(.)$, for $d \in \phi(c_j)$ on $q \in c_j$.
19:     $m_{j,d}$ = the solution of $F_{j,d}(.) = f$ on points $q$ that lie on the boundary of $\phi(c_j)$.
20: return $\cup m_{j,d}$, for $c_j = 1, \ldots, |C|, d \in \phi(c_j)$.
```

where (α_i, β_i) can be any of the points $p_i, p'_i, Q_R(y_i)$ or $Q_L(y_i)$, and $(x(q), y(q))$ can be $Q_R(y_i)$ or $Q_L(y_i)$. If both endpoints lie inside the region, or none of them do, we only have a constant independent of the coordinates of q and dependant only on the cell containing q; the rest of the segment lengths are the distances between the projection of the query point and the vertices of the cell.

Since we want to compute the boundary of the solution region, we need the maximum of the function. The maximum of a sum of square roots of quadratic polynomials, like this formula, are at their boundaries. Draw the length formula by elevating each point q on the plane by the length query of $Q(q)$. Since q is defined over c_j, which is a polygon, the boundary is a prism with c_j as its base. First, compute the intersection of $f = \sqrt{(x(q) - \alpha_i)^2 + (y(q) - \beta_i)^2}$ with the plane through an edge e of the boundary and parallel to the z-axis. An intersection of this type gives a parabola (See Figure 6). Note that $f_{i,d}$, the distance from each segment, is a parabola based on the equation $\sqrt{(x(q) - \alpha_i)^2 + (y(q) - \beta_i)^2} = f$. Since all such parabolas are upward and their centers are inside the cell, the maximum of their sum happens on the boundary.

The intersections of two parabolas can be computed in constant time. Add a new vertex on each of these intersections to the domain of the definition. So, it is enough to compute the value of $F_{j,d}$ at the vertices of the boundary, after

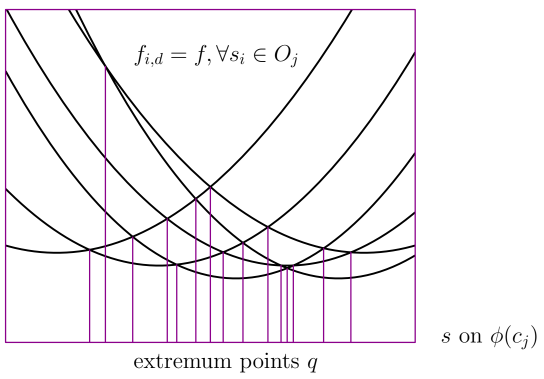

Fig. 6 The projection of the length function $F_{j,d}$ on the plane through each edge and parallel to the z-axis is a set of parabolas (Theorem 2).

dividing the domain of the function based on the conditions on the placement of q with respect to the segments.

Theorem 3 *The time complexity of Algorithm 3 is* $O(m^2n^3(\log m + \log n) + m^3n^2)$.

Proof Breaking the boundary based on the if statements requires $O(|Q|)$ time, since computing Q_R and Q_L take $O(|Q|)$ time and computing the intersection of two translations of Q also takes linear time since they are convex. Computing the maximum value by checking the boundary points takes linear time in terms of the number of vertices. Using Lemma 1, the complexity of the AQD is $O(m^2n^2)$, so the time complexity of the algorithm, except for computing AQD, is:

$$\sum_{c_j \in C} \sum_{s_i \in O_j} |Q| = \sum_{c_j \in C} |O_j||Q| = |Q| \sum_{c_j \in C} |O_j| = O(m)O(m^2n^2) = O(m^3n^2).$$

Using this bound and the one in Lemma 2, the total time complexity is proved to be $O(m^2n^3(\log n + \log m) + m^3n^2)$.

2.3 Length Query Using AQD

At query-time, we do a point location to find the cell of the AQD diagram that contains the representative point of B. Then, we compute the sum of lengths of parts of the intersecting segments that lies inside that cell, using the formula $F_j(q, \{p_i\}_{i=1}^{|O_j|}, \{p'_i\}_{i=1}^{|O_j|}, (d_1, \ldots, d_4))$ stored in that cell.

Theorem 4 *Given the output of Algorithm 3, computing a length query takes* $O(\log(nm) + km)$ *time, where* k *is the number of the edges of the curve that the query shape intersects.*

Algorithm 4 Length Query

```
Input: The AQD, the query $Q$
Output: The total sum of the lengths of the segments intersecting by $Q$
1: AQD = The output of Algorithm 3.
2: $c_j$ = Do a point location on AQD for finding $q$, the representative point of $Q$.
3: return $F_j(q, \{p_i\}_{i=1}^{|O_j|}, \{p'_i\}_{i=1}^{|O_j|}, (d_1, \ldots, d_4))$
```

Proof Computing the cell containing the representative point of the query shape requires solving a point-location to find the representative point of the query in the cells of AQD. Since the number of cells is $O(n^2m^2)$ as proved in Lemma 1, the point-location query takes $O(\log(n^2m^2)) = O(\log n + \log m)$ time. Given the AQD cell containing the representative point, we have the set of intersecting segments, and the exact length query can be computed by substituting the representative point of the query shape in the formula inside the cell. For each cell c_j, the complexity of formula F_j stored in c_j is $O(|O_j|m)$. When c_j is the cell containing the representative point of the query shape, $k = |O_j|$. So, the overall time complexity is $O(\log n + km)$.

2.4 Finding Popular Places Using AQD

One of the applications of AQD for convex polygons is computing popular places for such queries. We slightly modify Algorithm 1 (counting) for computing the AQD of a single polygonal curve to multiple polygonal curves by storing for each segment, the index of the curve that has it as one of its edges. Using the aforementioned diagram, Algorithm 5 solves the popular places problem.

Algorithm 5 Finding Popular Places Using AQD

```
Input: An integer $f$, a set of polygonal curves $P_1, \ldots, P_K$, a query shape $Q$
Output: The geometric locus of the representative points of shapes intersecting at least $f$
    input curves.
1: $R = \emptyset$
2: $S$ = the segments of each $P_i, i = 1, \ldots, K$, along with index $i$ of their corresponding $P_i$.
3: $AQD$ = The output of Algorithm 1 on $S$ and $Q$.
4: for each cell $c_j$ of AQD do
5:     if the number of distinct curves with segments in $c_j$ is at least $f$ then
6:         $R = R \cup \{j\}$
7: return the union of polygons that are the boundaries of cells $c_j$, for $j \in R$.
```

Lemma 4 *The boundaries of the polygons in the solution space of the popular places problem are a subset of the boundaries of the cells in the AQD of the curves.*

Proof Each cell of AQD has the same set of intersecting segments, so if one of the points in a cell is a popular place, then all of the points in that cell are popular places.

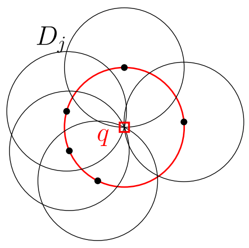

Fig. 7 The relation between the query size and the ply of the disks.

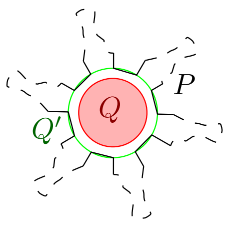

Fig. 8 Shapes Q and Q' have radii r and $r(1+\epsilon)$, respectively, but their corresponding length queries have an unbounded ratio.

Theorem 5 *Algorithm 5 solves the popular places problem in $O(m^2n^3(\log n+\log m))$ time.*

Proof Using Lemma 4, we know the solution is the union of a set of cells of AQD. Checking whether a cell intersects segments from at least f different curves can be done by checking the set of segments in each cell c_j, which takes linear time in $|O_j|$. Taking the union of two polygons takes time linear in the sum of the number of their vertices. So, it takes $O(n^2m^2)$ time. Using Lemma 2, we know the time complexity of computing an AQD is $O(m^2n^3(\log n+\log m))$, which dominates the time complexity of the rest of the steps in the algorithm.

3 An Algorithm for Length Query with Disk Queries

In Algorithm 1, we computed an AQD for a polygonal query. Here, we give an algorithm for disk queries. The Minkowski sum of a segment and a disk can be covered by a rectangle of the same length as the segment and the width equal to the diameter of the disk, and two copies of the disk, where each of which is centered at an endpoint of the segment, as depicted in Figure 9. This shape is called a hippodrome. For computing the arrangement of the Minkowski sum of the segments and the disks, since each cell can be identified by both circular arcs and straight line segments, we treat each case separately. Then we do a point location in both cases, and find the solution of each cell, and then provide a solution for the intersection of the cells. For computing the AQD of the arrangement of a set of rectangles we can use Algorithm 1.

3.1 A Lower Bound for Length Queries with Disks

We prove the length-query problem cannot be solved via discretization by points or approximation with another shape.

In Figure 8, an example of a curve is presented in which for a disk that is small enough, the length of the curve inside the query is zero, while a disk of slightly more radius has an unboundedly longer length. So, even the slightest modifications to the curve can change the output value drastically.

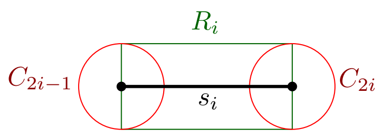

Fig. 9 The Minkowski sum of a segment s_i and a disk can be covered by a rectangle R_i and two disks C_{2i-1}, C_{2i} (Algorithm 6) that is a hippodrome.

Also, Figure 8 shows it can be the case that the maximum length happens for a disk whose representative point does not lie on the curve.

In Figure 7, we see an example of a query point and the set of disks containing that point. The output size k of a reporting windowing query is at most the maximum number of disks that cover a point, also known as the ply [15] of the disks. So, the time complexity of answering such a query is at least k.

3.2 Algorithm

In Figure 9, the Minkowski sum of a segment and a disk is presented. Such a shape can be converted into a rectangle R_i and two half-disks.

To simplify the problem, we solve the reporting problem on the disks and on the rectangles separately, and then take their union. Since the sets returned

Algorithm 6 Length Query for Disk Queries: Preprocessing

```
Input: A set of segments $S$, a constant $r > 0$
Output: Two data structures for point location
1: for each segment $s_i \in S$ do
2:     $P_i$ = compute the Minkowski sum of a disk of radius $r$ and $s_i$
3:     divide $P_i$ into a rectangle $R_i$ and two disks $C_{2i-1}, C_{2i}$.
4: Compute a data structure for point location for $R_i, i = 1, \dots, |S|$ using Algorithm 1.
5: Compute a data structure for point location for $C_i, i = 1, \dots, 2|S|$ using Algorithm 8.
```

by each of these queries are part of the solution, the total space requirement of the query remains unchanged.

Algorithm 7 Length Query for Disk Queries: Query

```
Input: A set of segments $S$, a disk $Q$ of radius $r > 0$
Output: The sum the lengths of the segments inside $Q$
1: $O_1$ = the set of indices from the reporting query for $R_i, i = 1, \dots, |S|$ using Algorithm 2.
2: $O_2$ = the set of indices from the reporting query for $C_i, i = 1, \dots, 2|S|$ using Algorithm 10.
3: return the sum the lengths of the segments from set $O_1 \cup O_2$ inside $Q$.
```

3.2.1 A Data Structure for Point Location in a Set of Congruent Disks

In this section we propose a data structure for point location in congruent disks which is described in Algorithms 8 and 9. Let points V_i be the centers of the disks D_i, for $i = 1, \dots, n$.

Algorithm 8 Point-Location in Congruent Disks: Preprocess

```
Input: A set of disks $D_1, \dots, D_n$ of radius $r$
Output: A point-location data structure for congruent disks
1: Build the intersection graph $G$ of $D_i$, for $i = 1, \dots, n$.
2: $L_i$ = the list of intervals of intersections between $D_i$ and $D_j$, for disks $D_j$ adjacent to
     $D_i$ in $G$, sorted on the first met point in the clockwise order starting from the $x$-axis
     along with the index $j$.
3: $I_i$ = the intersection of $D_i$ with the tangents from $V_i$ to $D_j$, for disks $D_j$ adjacent to
     $D_i$ in $G$, if $r < d(V_i, V_j) \leq \sqrt{2}r$, sorted on the first met point in the clockwise order
     starting from the $x$-axis.
4: $L'_i$ = break the intervals of $L_i$ on the points of $I_i$.
5: $T_i$ = build a segment tree on each set $L_i$, for $i = 1, \cdots, n$.
6: $Y_{i,v}$ = store the order of the disks in each leaf $v$ of $T_i$ based on their distances from $V_i$
     on both sides of the cone at $V_i$ facing arc $v$, along with their disk indices.
7: return $T_i$ augmented by $Y_{i,v}$, $\forall i = 1, \cdots, n$, $\forall v \in L_i$
```

In the data structure of Algorithm 8, each set $Y_{i,v}$ consists of a set of arcs, whose circles may or may not contain V_i (See Figure 12 for an example). We define a comparison function $\psi(q, V_i, D_j)$ as the comparison function to check whether a point q lies before the arc of the disk D_j inside the cone with apex V_i:

$$\psi(q, V_i, D_j) = \begin{cases} \text{Is } q \text{ inside } D_j? & \text{if } V_i \in D_j \\ \text{Is } q \text{ outside } D_j? & \text{otherwise} \end{cases}.$$

Algorithm 9 uses $\psi(q, V_i, D_j)$ to do a binary search at Line 5.

Algorithm 9 Point-Location in Congruent Disks: Query

```
Input: A point-location data structure (output of Algorithm 8), $q$: a query point
Output: The face of the AQD that contains $q$
1: $V_i$ = the nearest neighbor of $q$ among $V_1, \cdots, V_n$.
2: if $d(V_i, q) > r$ then return the external face
3: $q'$ = the intersection of the ray from $V_i$ to $q$ with the boundary of $D_i$.
4: $v$ = query $T_i$ for $q'$.
5: $y$ = binary search on $Y_v$ to find the face containing $q$ using $\psi(q, V_i, D_j)$ for $D_j \in Y_v$.
6: return $y'$
```

Here, we give an example of our algorithms (Algorithms 8 and 9) for point-location on a set of congruent disks.

Preprocessing (Figure 10)

1. The nearest neighbor of q is b, so C_i is the disk centered at b. Based on the neighbors of C_i in the intersection graph of the disks, the set of intersecting circles is $\{a, c, d\}$.
2. Compute the part of the arc of the current circle C_i that falls inside an intersecting circle C_j, and intersection points for all intersecting disks and sort them from an arbitrary point on the boundary. The lists of intervals of these disks are $L_b = \langle(1, 3), (3, 5), (4, 8), (7, 1)\rangle$. Each point on the boundary of a circle can be inside the intersection of several other circles, therefore, the intervals are not disjoint.
3. The tangents from V_i to the disk have touching points inside C_i only for the disk centered at a. So, $I_b = \langle 2, 6\rangle$.
4. Using L_b and I_b, we can compute $L'_b = \langle(1, 2), (2, 3), (3, 5), (4, 6), (6, 8), (7, 1)\rangle$.
5. The leaves of T_i are $(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 1)$. Each of them stores a set of arcs, sorted on the distances of their intersections with the sides of the smallest cone containing them to V_i.

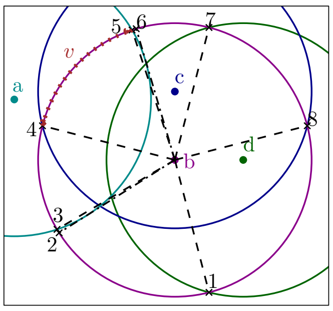

Fig. 10 Preprocessing (Algorithm 8)

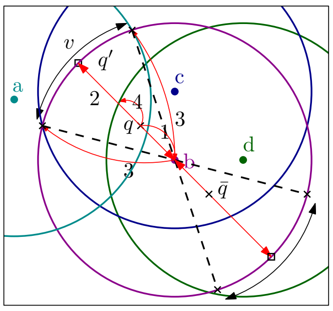

Fig. 11 Query (Algorithm 9)

Query (Figure 11) Assume a, b, c, d are the centers of circles C_a, C_b, C_c, C_d and q is the query point.

1. The nearest neighbor of q is b.
2. Connect q to the boundary of C_b.
3. Query the segment tree of the disjoint arcs on the boundary of C_b to find the dashed lines (interval v).
4. $Y_{i,v} = \langle a, d\rangle$.
5. Binary search on the intersection of circles assigned to interval v, on each of the dashed lines. Using the results of the comparison function $\psi(q, b, C_a) = true, \psi(q, b, C_d) = false$, the face containing q is determined.

Theorem 6 *Algorithm 9, with Algorithm 8 as preprocessing, solves the point location problem in a set of congruent disks.*

Proof If the output-size is non-zero, the nearest neighbor of q in set of centers of D_i, for $i = 1, \ldots, n$, is a part of the solution. Assume this disk is D_j. In that case, by definition, the distance between q and the center of D_j is less than the radius, so all the disks containing q also intersect with D_j. Since L_j is the list of the maximal arcs of D_j that are inside an intersecting disk D_i, the line through V_i and q intersects with all the disks that can contain q.

The radii of the disks are equal, so given two points, namely the endpoints of an interval in L_j, there are at most two disks that pass through those points. One of them contains the point V_j, which in this case can only be D_j, and the other one does not, which is D_i.

The leaves of segment trees contain disjoint intervals, called elementary intervals. This property guarantees the intersections between no two disks intersecting D_j is inside an elementary interval, when mapped to the closest point on the boundary of D_j. So, for each point q, the set of intersecting disks can be uniquely determined using L_j: the subset that lies inside the leaf of T_i containing q(See Figure 12). However, there is a case when a disk intersects only one side of the cone twice (See Figure 13). We handle this case by adding the points I_i to the set L_i in the algorithm, which gives the set L'_j. Consider the ordering of arcs in the cone at V_i facing an interval in a set L_i of Line 2 on both sides of a cone. These orderings are inconsistent if the tangent from V_i to the arc of the intersecting disk falls inside that interval (See Figure 13). The tangent line from the center of the disk C_i to any other disk C_j falls inside C_i if

$$2r^2 = d(V_i, V_j)^2 \Rightarrow d(V_i, V_j) = r\sqrt{2},$$

where V_i, V_j are the centers of C_i and C_j, and r is the radii of the disks. If we draw a tangent from V_i to D_j, since the disks have the same radius, the tangent point lies inside D_i. We handle this case by adding the points I_i to the set $Y_{i,v}$ in the algorithm, where information about other cones made by tangents to such disks is also stored.

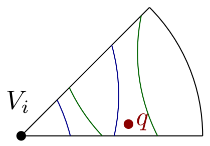

Fig. 12 The set of interior-disjoint arcs $Y_{v,i}$ in Algorithm 8, equivalently, Y in Algorithm 9.

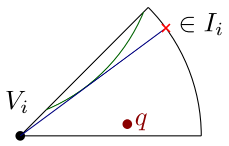

Fig. 13 Two cones through a point of I_i in $Y_{v,i}$ in Algorithm 8, equivalently, Y in Algorithm 9.

As a result, there is a monotone order among the set of arcs from the disks corresponding to L'_j in the leaf containing the projection of q onto D_j. Storing the order can be done by sorting the intersections with the cone constructed by connecting the center of D_j to the endpoints of the first and last endpoint of L'_j in clockwise order, starting from the x-axis. In the binary search on the set of disks, the checks for q being on the same side of the arc as V_i are done via circle inclusions/exclusions. This gives an arc adjacent to the face containing

q, depending on the end condition of the binary search, which can be used to find the face containing q.

Theorem 7 *The time complexity of Algorithm 8 is $O(n^3 \log n)$.*

Proof The time complexity of computing the intersection graph of n disks is $O(n^2)$. Finding the nearest neighbor requires building a Voronoi diagram, which takes $O(n \log n)$ preprocessing time. Building a segment tree on each disk requires $O(n \log n)$ time, so the total time required for building segments trees for all disks is $O(n^2 \log n)$. Each elementary set of each segment tree can have a size at most n. Sorting each list takes $O(n \log n)$ time. The total number of leaves of a tree is also $O(n)$, so the total time is $O(n^3 \log n)$.

Theorem 8 *The time complexity of Algorithm 9 is $O(\log n)$.*

Proof The nearest neighbor query using a Voronoi diagram takes $O(\log n)$ time. A query in the segment tree takes $O(\log n)$ time. Finding the face in the sorted list of arcs corresponding to an elementary interval of the segment tree takes $O(\log n)$ time, using binary search.

3.2.2 Reporting Intersected Disks

The point location algorithms of Section 3.2.1 can be extended to intersection reporting queries by keeping the set of intersected disks in each face at preprocessing time.

Algorithm 10 Point-Location in Congruent Disks: Reporting Query

```
Input: $G$: a point-location data structure for disks, $q$: a query point
Output: The set of disks that contain $q$
1: $Y_{v,i}$ = run Algorithm 9 on $q, G$.
2: $\bar{q}$ = the reflection of $q$ with respect to $V_i$
3: $Y_{v',i}$= run Algorithm 9 on $q', G$
4: $y$ = the members of $Y_{v,i}$ that are between $q$ and $V_i$, except repeated items (two arcs
   from the same disk).
5: $y' = Y_{v',i} \setminus Y_{v,i}$.
6: return $\{V_i\} \cup y \cup y'$
```

Theorem 9 *The time complexity of Algorithm 10 is $O(\log n + k)$, with Algorithm 8 as preprocessing, where k is the number of segments intersecting Q.*

Proof Based on Theorem 8, the time complexity of finding the nodes that contain q is $O(\log n)$. Reporting the list of intersecting segments takes $O(k)$ time, since reporting the values before q in the sorted list takes time proportional to the number of such values.

4 Disk Queries And c-Packed Curves

We can solve the length query problem on a class of curves called c-packed curves for disk queries with arbitrary radii. Interestingly, our method gives a constant factor approximation for computing the value c for which a curve is c-packed, if such a constant c exists.

4.1 Application: Approximating The c-Packedness of A Curve

The notion of c-packed curves has been extensively used in the literature, however, there are no algorithms for computing the values of c for a given curve. Using AQD algorithm, it is possible to find an approximation of the minimum value for c so that the input curve is c-packed.

Since any curve is L-packed for L equal to the length of the curve, then we focus on the interval $[0, L]$. The closest points among the points of P is denoted by e^*, and it can be computed by taking the minimum distances between non-intersecting edges of the curve. For a given r, we binary search on $[e^*, L]$ to find a $(1+\epsilon)$ approximation of r, for which the curve P has length at most cr inside each disk of radius r. The algorithm builds an AQD for a disk of radius r as the input shape and computes the maximum query value among all cells, based on the formula in each cell.

Algorithm 11 Approximating c-Packedness

```
Input: A curve $P$, a constant $\epsilon > 0$
Output: The minimum $c$ for which $P$ is $c$-packed
1: $c = 0, \epsilon \leftarrow \epsilon/2$
2: $L$ = compute the length of the curve $P$.
3: $e^* = \min_{p,q \in P} \|p - q\|$
4: for $i = e^*, \ldots, \log_{1+\epsilon} L$ do
5:     $r = (1+\epsilon)^i$
6:     $Q$=a regular polygon with $\lceil \frac{\pi}{\sqrt{\epsilon}} \rceil$ vertices
7:     cost=Algorithm 3 for $Q$
8:     $c = \max\{c, \frac{cost}{r}(1+\epsilon)\}$
9: return $c$
```

Lemma 5 *If a curve has length at most cr inside disk queries of radius r centered at a point on the curve, for all $r > 0$, then it is $(2c)$-packed.*

Proof Disks whose centers have distance more than r from the curve do not intersect with it, so, their length query is zero. Based on the statement of the lemma, the disks of radius r with their centers on the curve have length at most cr, for any $r > 0$. Any disk of radius r that intersects the curve can be covered by a disk of radius $2r$ with its center at a point inside the intersection of the disks. The length of the curve inside the disk of radius $2r$ is at most $2cr$. This is an upper-bound on the length of the curve inside the smaller disk. So, the curve is at most $2c$-packed. □

Theorem 10 *Algorithm 11 computes a $(2+\epsilon)$-approximation of the minimum c for which the curve P is c-packed, in $O(\frac{\log L/e^*}{\epsilon})$ time, where L is the length of P and e^* is the distance between the closest points on P, if $c = O(1)$.*

Proof We prove the theorem for disks whose centers lie on the input curve, and use Lemma 5 to extend it to any disk, at the cost of a factor 2 in the approximation factor.

For $r \geq L$, the translation of the query disk whose center lies on the first vertex of the polygonal curve P, contains the whole curve. Any point other than the endpoints covers at least as much length, so in this case the optimal solution is achieved.

For $r \leq e^*$, the set of intersecting segments for any disk of radius r is the same as a disk of radius e^*.

Now we discuss $e^* < r < L$. The length of the curve inside a disk Q can be bounded by the length of the curve inside the largest inscribed regular k-gon in Q and the length of the curve inside the smallest circumscribed regular k-gon about Q. We set k such that the radius of the circumscribing circle of the regular k-gon be at most $1+\epsilon$ times the radius of its circumscribed circle. The proof is similar to the proof of ϵ-kernel. Let $\theta = \frac{2\pi}{k}$. The ratio between the radius of the circumscribing circle and the circumscribed circle is:

$$\frac{r}{r\cos(\theta/2)} \approx \frac{1}{1-\frac{\theta^2}{4}} = 1 + \frac{\frac{\theta^2}{4}}{1-\frac{\theta^2}{4}} \approx 1 + \frac{\theta^2}{4} = 1+\epsilon,$$

for $\theta \to 0$. So,

$$\frac{\theta^2}{4} = \epsilon \Rightarrow \theta = \frac{2\pi}{k} = 2\sqrt{\epsilon} \Rightarrow k = \frac{\pi}{\epsilon}.$$

As shown in Figure 8, these bounds are not within a bounded ratio of each other. We bound the cost by the cost of the inscribed k-gon in the disk. So, for a length query of cost ℓ_X, where X is the query shape,

$$\ell_{\text{k-gon}} \leq \ell_{\text{disk}} \leq \ell_{\text{k-gon}}(1+\epsilon).$$

This ratio holds for the length query as well, since the length of the curve inside the query shape is at least r and after adding $r\epsilon$ to its length, it still gives a $(1+\epsilon)$-approximation, so:

$$\ell_{\text{disk}(r)} \leq \ell_{\text{disk}(r(1+\epsilon))} \leq (1+\epsilon)\ell_{\text{disk}(r)},$$

where disk(r) denotes the disk query of radius r.

The approximation of c is given by $\frac{\ell_{\text{k-gon}}}{r}$, the length query inside the k-gon divided by the radius of the query disk circumscribing about the k-gon. The optimal c is the maximum of the length query to r, for all $r > 0$, so testing values $r = (1+\epsilon)^i$ instead of all values r gives a $(1+\epsilon)$-approximation. The overall approximation factor is the product of the approximation factors, which is $(1+\epsilon)^2 \approx 1 + 2\epsilon$. □

4.2 An Algorithm for c-Packed Curves

We give a $2c$-approximation algorithm for computing length queries with a disk of radius r as the query on c-packed curves. It is enough to compute the distance between the center of the query disk and the nearest segment of the curve. This reduces the problem to an instance of the nearest segment problem for a point query, which can be solved in $O(\log n)$ time using a Voronoi diagram of line segments [9].

Algorithm 12 Preprocessing

```
Input: A curve $P$ with edges $\{e_1, \cdots, e_{n-1}\}$
Output: A constant $c$, a segments' Voronoi diagram $D$
 1: Build a Voronoi diagram $D$ on the set of segments $\{e_1, \ldots, e_{n-1}\}$.
 2: Compute the $c$-packedness of $P$ using Algorithm 11.
 3: return $D$,$c$
```

If the constant c for which the curve is c-packed is unknown, we use Algorithm 12 to approximate c before running Algorithm 13. Algorithm 13 tests two segments if the nearest point of the curve is a vertex, and one if it is an edge.

Algorithm 13 Query

```
Input: A segments' Voronoi diagram $D$ of a $c$-packed curve, a constant $c$, a disk query $Q$
    with representative point $q$ and radius $r$
Output: The length of the curve inside $Q$
 1: $S$ = the set of cells of $D$ returned by the point-location on point $q$.
 2: for $s \in S$ do
 3:     $d$ = the distance between the segment $s$ corresponding to $m$ and $q$.
 4:     sum=0
 5:     $\beta$ = the angle between the line containing $s$ and the line through $q$ and the closest
    point of $s$ to $q$
 6:     if $d < r$ then
 7:         if $\beta \in [0, \cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}})]$ then
 8:             $sum = sum + |r - d|$
 9:         else if $\beta \in (\cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}}, \frac{\pi}{2}]$ then
10:             $sum = sum + \sqrt{r^2 - d^2\cos^2(\beta)}$
11:         else
12:             $sum = sum + r$
13: return $sum$
```

Theorem 11 *Algorithm 13 gives a $2c$-approximation for the length queries of disks of radius $r \leq \min_{s \in D} \|s\|$.*

Proof Based on the definition of c-packed curves, for queries whose centers lie on the curve, r is a c-approximation, since the length of the curve inside the query shape is at least r and at most cr.

We compute a lower bound on the length query of disks whose centers (q) is not on the curve. Since for c-packed curves, there is an upper bound cr on the length of the query inside the ball of radius r, for any $r > 0$, we only need to give a lower bound for the length of the curve inside the ball to prove an approximation factor.

Let d be the distance between q, the center of the query disk Q, and its closest point on the curve. Assume the angle between the line passing through the centers of the disks and the segment of the curve is β, and the angle of the arc inside the intersection of the disks is α. For all $d > r$, the intersection is 0. So, we discuss $d \in [0, r)$.

$$\frac{d}{r} = \cos(\frac{\alpha}{2})$$

So, the perpendicular chord length is:

$$r\sin(\frac{\alpha}{2}) = r\sqrt{1 - \frac{d^2}{r^2}}.$$

To add the effect of β, it is enough to replace d with $d' = d\cos(\beta)$.

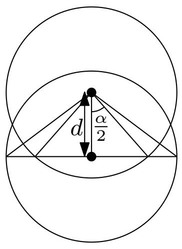

Fig. 14 The figure of Theorem 11.

Now we compute the length of the perpendicular from the center of the disk to the chord. For $\beta \in [\frac{\pi}{2}, \pi]$, the length of the perpendicular is at least r. So, we compute the length of the perpendicular for $\beta \in [0, \frac{\pi}{2}]$, which is $(r - d)\cos(\beta)$.

The maximum of these lower bounds gives a stronger lower bound on the length of the curve inside the disk:

$$\max_{\beta}(r\sqrt{1 - \frac{d^2}{r^2}\cos^2(\beta)}, (r - d)\cos(\beta)).$$

Since the functions are monotone, the maximum happens at an endpoint of the intervals:

$$\begin{aligned} r\sqrt{1-\frac{d^2}{r^2}\cos^2(\beta)} &= (r-d)\cos(\beta) \Leftrightarrow \\ r^2-d^2\cos^2(\beta) &= (r^2-2rd+d^2)\cos^2(\beta) \Leftrightarrow \\ \cos^2(\beta) &= \frac{r^2}{r^2-2rd+2d^2} \Leftrightarrow \\ \cos(\beta) &= \frac{r}{\sqrt{r^2-2rd+2d^2}}. \end{aligned}$$

So the maximum of the lower bounds on the length are:

$$\begin{cases} |r-d| & \beta \in [0, \cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}})] \\ \sqrt{r^2-d^2\cos^2(\beta)} & \beta \in (\cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}}), \frac{\pi}{2}] \end{cases}.$$

The bound $r \leq \min_{s\in D}\|s\|$ guarantees each part of the curve inside the query shape has at most two segments. Since we are bounding the adjacent segments separately, and the closest point is within distance d of the center, the remaining length is $r-d$. A disk of diameter $r-d$ contains this segment entirely, so the upper bound is $c(r-d)$. We use the upper-bound cr for c-packed curves which yields the following ratios:

$$\begin{cases} \frac{cr}{|r-d|} & \beta \in [0, \cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}})] \\ \frac{cr}{\sqrt{r^2-d^2\cos^2(\beta)}} & \beta \in (\cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}}), \frac{\pi}{2}] \end{cases}.$$

For $d \in [r-\delta, r]$, for a constant $\delta > 0$ to be determined later, we have $1-\frac{\delta}{r} \leq d/r \leq 1$, so assuming $\delta \leq r/2$:

$$\frac{r^2}{r^2-2rd+2d^2} = \frac{1}{(1-d/r)^2+(d/r)^2} \leq \frac{1}{(\delta/r)^2+(1-\delta/r)^2} \Rightarrow$$

$$\sqrt{\frac{r^2}{r^2-2rd+2d^2}} \leq \cos(\beta) \Rightarrow \beta \leq \cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}}).$$

So, the first case of the approximation factor is used, which for $\delta = r\epsilon$ gives the bound:

$$\frac{cr}{|r-d|} = \frac{cr}{\delta} = \frac{cr}{r\epsilon} = \frac{c}{\epsilon}.$$

For $d \in (0, r\epsilon)$, we use a relaxed bound:

$$\leq \begin{cases} \frac{cr}{|r-d|} & \beta \in [0, \cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}})] \\ \frac{cr}{\sqrt{r^2-d^2}} & \beta \in (\cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}}), \frac{\pi}{2}] \end{cases}$$

Using the assumption $0 \leq d \leq r\epsilon$:

$$0 \leq d^2 \leq r^2\epsilon^2 \Rightarrow r^2-r^2\epsilon^2 \leq r^2-d^2 \leq r^2 \Rightarrow r\sqrt{1-\epsilon^2} \leq \sqrt{r^2-d^2} \leq r.$$

Substituting these values in the previous ratios gives the bound on the approximation factor:

$$\leq \begin{cases} \frac{cr}{r(1-\epsilon)} & \beta \in [0, \cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}})] \\ \frac{cr}{r\sqrt{1-\epsilon^2}} & \beta \in (\cos^{-1}(\sqrt{\frac{r^2}{r^2-2rd+2d^2}}), \frac{\pi}{2}] \end{cases}$$

The approximation factor in this case is $\frac{c}{1-\epsilon}$, since

$$\begin{aligned} \frac{c}{1-\epsilon} \geq \frac{c}{\sqrt{1-\epsilon^2}} &\Leftrightarrow \sqrt{1-\epsilon^2} \geq 1-\epsilon \\ &\Leftrightarrow 1-\epsilon^2 \geq (1-\epsilon)^2 = 1-2\epsilon+\epsilon^2 \\ &\Leftrightarrow 0 \geq 2\epsilon^2 - 2\epsilon = 2\epsilon(1-\epsilon). \end{aligned}$$

The maximum of these values gives the approximation factor of the problem, which is minimum for $\epsilon = 1/2$:

$$\min_{\epsilon \in [0,0.5]} \max(\frac{c}{\epsilon}, \frac{c}{1-\epsilon}) = 2c.$$

5 MapReduce Algorithms

Computing a length query for a single input curve can be done in $O(1)$ rounds in MapReduce by simply distributing the line segments of the curve among machines and sending the query shape to all machines, and taking their union. Note that in the MapReduce model, the local time complexity of each machine just needs to be polynomial. Consequently, we also do not consider the time complexity of the local computations in each machine.

However, for finding popular places using length queries, one needs to solve an infinite number of instances of length query problem: one per point of the plane. Instead, we use the preprocessing algorithm for length query (Algorithm 3 for polygonal queries and Algorithm 8 for disk queries) to find a subset of points that maximize F_j for $j = 1, \ldots, k$. Then, we use these points as vertices of the boundary of the solution space of the popular places problem. The edges can be then computed using these vertices.

5.1 A MapReduce Algorithm for Finding The Intersections Among x-Monotone Polygonal Curves

In this section, we introduce a MapReduce algorithm for computing the set of intersection points in an arrangement of a set of K x-monotone curves. We will use this result to compute the solution to the problem of finding popular places in Section 5.2. Observe that the number of segments intersecting a vertical segment is K, if the segments are the edges of a set of K x-monotone curves.

Lemma 6 *On a given set of K x-monotone curves of total complexity n, the total number of intersections is at most $O(nK)$.*

Proof First project all the curves on the x-axis. The number of the intersecting intervals is $O(nK)$, since n intervals each repeated at most K times. Since the x-values are fixed, assuming no axis-aligned segments exist in the input, the y-values are also distinct. This concludes the proof.

Note that because of space limitations of MapReduce models, the maximum number of segment intersections can be $O(n)$ in MPC and $O(M\ell) = o(n^2)$ in MRC.

Algorithm 14 Segments Intersections of x-Monotone Curves in MapReduce

```
Input: A set of segments $S = \{s_1, \ldots, s_n\}$ from $K$ $x$-monotone curves
Output: The intersection points between the segments of $S$
1: Sort and distribute the endpoints of segments in $S$ based on their $x$ values, such that
   each machine has $\frac{n}{\ell}$ values.
2: $X$ = the smallest $x$-value in each machine.
3: Send $X$ to all machines using a parallel prefix algorithm.
4: Break each segment at points of $X$ to get the set $S' = \{s'_1, \ldots, s'_{n\ell}\}$.
5: Partition $S'$ based on $X$ to sets $S'_i$ for $i = 1, \ldots, n\ell$
6: Distribute the elements of $S'$ to the machines
7: Locally compute the intersections in each machine.
```

Theorem 12 *Algorithm 14 takes $O(\log_M n)$ rounds and $O(nk)$ total space, using ℓ machines of memory $M, M \geq \frac{nK}{\ell}$ in MRC, if $K = o(M)$.*

Proof Sorting n items in MRC takes $O(\log_M n)$ rounds, so sorting the endpoints in the first step takes $O(\log_M n)$ rounds. Sending X to all machines using parallel prefix takes $O(\log_M |X|)$ rounds. Since $|X| = \ell$, the round complexity is $O(\log_M \ell) = O(\log_M n)$ rounds. The total amount of memory required should be at least as much as the size of S'. By the construction of S', we know that $|S'| = |X|K + n = \ell K + n$. Also, the set of segments from S' in each machine can be at most $\frac{n}{\ell}K$, so the memory of each machine should be at least $\frac{nK}{\ell}$.

5.2 A MapReduce Algorithm for Finding Popular Places

In Section 2.4, we proved the edges and vertices of the solution space of the popular places are a subset of the edges of AQD. Algorithm 15 finds the set of vertices of the solution space of the popular places problem. The max-length popular places can be solved similarly by replacing the local algorithm with one of our sequential algorithms, attaching the solutions on the boundary vertices of the partitions and summing up the values at boundary vertices of the partitions.

Algorithm 15 Popular Places in MapReduce

```
Input: A set of segments $S = \{s_1, \ldots, s_n\}$ from $K$ $x$-monotone curves, a query $Q$, an
    integer $f$
Output: The popular places region as a graph
 1: Compute the Minkowski sum of each $s_i$ and $Q$, and add its edges to $W$
 2: Run Algorithm 14 on $W$ to find the set of intersections $I$
 3: Compute the edges between the vertices in set $I$ locally.
 4: Compute the complexity of each face locally by a sweep-line algorithm.
 5: return the faces with list sizes greater than or equal to $f$
```

Theorem 13 *Algorithm 15 solves the popular places problem for K x-monotone curves in $O(\log_M nm)$ rounds and $O(nmK)$ space in MRC, if $K = O(M)$.*

Proof Computing the Minkowski sum is done locally, so it does not require an extra round. Using Theorem 12, computing the intersections takes $O(\log_M |W|) = O(\log_M(nm))$ rounds and $O(|W|K) = O(nmK)$ space. Locally computing the edges between the members of I does not require a separate round, however, sending the vertices events in the sweep-line algorithm at the x-values that separate data in different machines requires a round. Consequently, each machine can compute the complexity of its involved faces locally, even if only some edges of a cell are distributed to this machine. The overall round complexity is therefore $O(\log_M nm)$ and the overall space complexity is $O(nmk)$.

6 Experimental Results

To evaluate the performance of our algorithm in practice, we do our computational tests on a real trajectory data-set. The sequential experimental test is performed on a Core (TM) i2CPU and 2GB RAM computer with Windows 7 operating system.

We have implemented our algorithms in C++ with Visual Studio 2013. Figures 16 and 17 were drawn with RapidMiner Studio Professional 7.3.001. The other figures of the current section were drawn by LibreOffice 2019.

Our data-set includes real trajectories tracked with a GPS and collected in the Geolife project (Microsoft Research Asia), which contains the movements data of 178 users over 30 cities of China in a period of over four years. In a broader view, this data-set contains 17,621 trajectories with a total duration of 48,203 hours and a total distance of 1,251,654 kilometers. Almost 91 percent of the data is collected every 1 to 5 seconds or every 5 to 10 meters per point [34, 35, 36]. Since the data-set was very huge, we have applied our implementation on the data of the first user of the data-set, in which there exist 14,186 sample points.

6.1 Sequential Algorithm Experimental Results

We have implemented Algorithm 13 on the bounded-size input that is varied from 12 KiloBytes (KB) to 55 KB. In all the tests, the shape of the query is a disk. To analyze the running time of the algorithm, we have repeated our test on four different input size categories. This is carried out by dividing the input data by their size to the following categories: 12-25 KB, 25-35 KB, 35-45 KB, and 45-55 KB. Also each test is repeated for four different values of r that are 0.0125, 0.025, 0.035 and 0.045. The summary of the results of Algorithm 13 is shown in Table 3. Also, with three different values of f as a threshold of being popular, we have counted the number of popular places of each case. The spent time and the complexity of the AQD are the averages of the data in the mentioned size-range. In Figure 15, we have depicted the number of popular places of each r for each size-range category of the input.

In the reported times in Table 3 we have excluded the construction time of the Voronoi Diagrams.

As a result, on the range 12 KB to 55 KB, the spent time of the algorithm is completely reasonable. However, for the larger size, we needed to implement the MapReduce algorithms to achieve such a reasonable time.

6.2 MapReduce Experimental Results

For dealing with the problem of popular places on a big data-set, we have introduced a MapReduce algorithm. To be able to run our algorithm on the input curve, we needed an extra condition that the total number of the curves that are passing through a vertical line is at most K. To meet this criterion, we performed some preprocessing on the input curve.

Since the sampled points in several cases were very close, before running our algorithms, we have rounded almost the same data by rounding the sampled points (to achieve the accuracy of five digits for floating points), which we call them aggregated points. We have considered an extra parameter that is the weight for each aggregated instance of the data, which is the number of times that segment was traversed. With this preprocessing, we converted the input curves to a fewer set of x-monotone curves with weighted edges. For a total of 14,186 sample points, we had 57,633 and 58,183 aggregated segments for square queries of radii 0.5 and 0.025, with a total number of 181,405 and 63,392 segments in partitions to be sent to the machines, for the same radii.

The input is drawn in Figure 16 and the outputs of our algorithm are drawn in Figures 17 and 18. In Figures 17 and 18 the boundaries of the solution space of the popular places problem are drawn, in which the colors represent the popularity of that point. Some jitter was added to the figure to make the colors visible.

In our experiments, 277 machines each with 850 aggregated segments have been used. The union of the solutions had sizes 63,303 and 63,392 for radii 0.5 and 0.025, respectively.

Input size (KB)	r	$f = 0.015$	$f = 0.02$	$f = 0.025$	Time (s)	AQD size
12-25	0.0125	9182	2744	26	1.0249	532
	0.025	5393	2175	842	0.8679	532
	0.035	5358	2162	832	1.0317	532
	0.045	5383	2176	834	0.5964	359
25-35	0.0125	14346	12789	11034	3.3583	1130
	0.025	14544	13085	11502	3.0981	1130
	0.035	14540	13071	11510	3.0788	1130
	0.045	14514	13063	11494	3.0073	1124
35-45	0.0125	36968	32597	29077	6.0853	1762
	0.025	37517	34553	31710	5.7762	1737
	0.035	37222	34262	31456	5.9387	1751
	0.045	37484	34581	31738	5.5676	1753
45-55	0.0125	20909	19448	17584	10.17	2113
	0.025	21141	19913	18544	9.0423	2135
	0.035	21169	19923	18533	8.923	2136
	0.045	21116	19877	18522	9.1894	2114

Table 3 Experimental results of the sequential algorithm of the popular places problem.

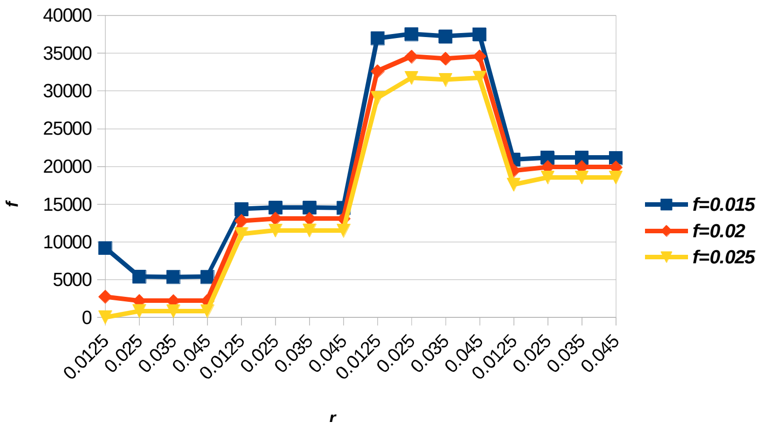

Fig. 15 The number of popular places of each r value for each category of the input. The plotting of the categories on the x-axis is ordered by increasing the category's size

7 Conclusions And Open Problems

While many data structures in computational geometry focus on counting and reporting queries, we propose a data structure for a more complicated query: computing the sum of the lengths of segments inside a query region. Prior to our work, one had to first solve the reporting problem, and then compute the sum of lengths on the results. For length queries with polygonal shape, we get a logarithmic query time. Also, we give a constant factor approximation for disk queries with logarithmic query time on c-packed curves. This gives the first algorithm for computing the minimum value c for which a curve is c-packed, if such a constant value c exists. Proving similar results for pseudolines is also interesting, where point location is more challenging.

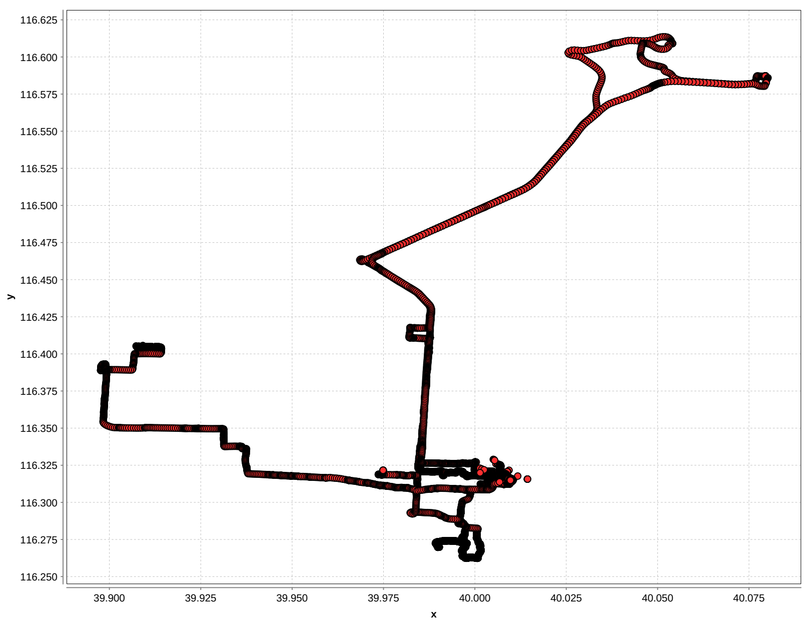

Fig. 16 The input curve S.

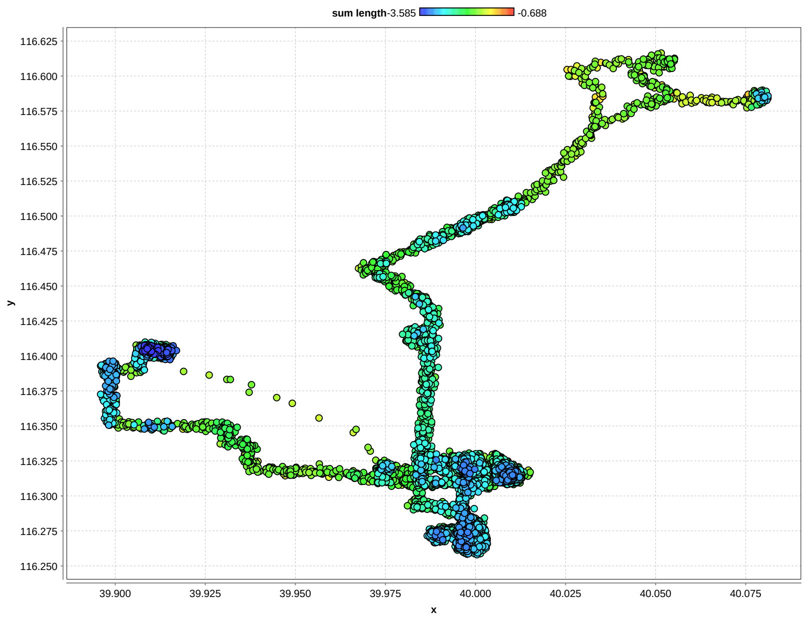

Fig. 17 The middle part of the boundary of the popular places of S computed by our algorithm for square queries of size 0.5×0.5. The color of each point represents its popularity.

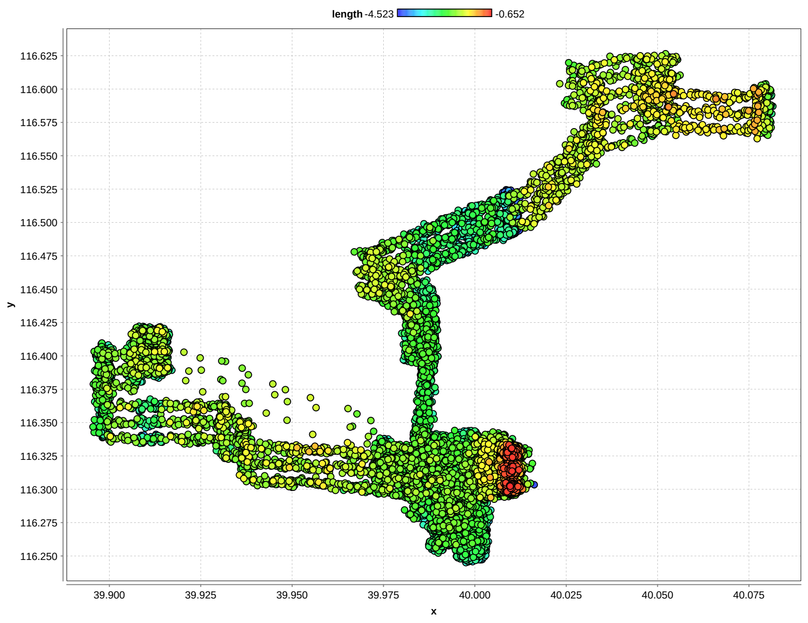

Fig. 18 The boundary of the popular places of S computed by our algorithm for square queries of size 0.025×0.025. The color of each point represents its popularity.

We extend our results to MapReduce and give an algorithm for computing segment intersections when the input curves are x-monotone and if the output fits in the total available memory. Using this algorithm, we solve the popular places problem for x-monotone curves. Solving the problem in MapReduce for general polygonal curves or an arbitrary set of segments remains open.

Acknowledgment

The length query problem was an open problem in the Winter School on Computational Geometry 2014 at Amirkabir University of Technology.

References

1. Agarwal PK, Har-Peled S, Varadarajan KR (2004) Approximating extent measures of points. Journal of the ACM (JACM) 51(4):606–635
2. Arya S, Malamatos T, Mount DM (2007) A simple entropy-based algorithm for planar point location. ACM Transactions on Algorithms (TALG) 3(2):17
3. Barequet G, Dickerson M, Pau P (1997) Translating a convex polygon to contain a maximum number of points. Computational Geometry 8(4):167–179

4. Beame P, Koutris P, Suciu D (2013) Communication steps for parallel query processing. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems, ACM, pp 273–284
5. Benkert M, Djordjevic B, Gudmundsson J, Wolle T (2010) Finding popular places. International Journal of Computational Geometry & Applications 20(01):19–42
6. Benson RV (1966) Euclidean geometry and convexity. McGraw-Hill
7. Bringmann K (2014) Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless seth fails. In: 2014 IEEE 55th Annual Symposium on Foundations of Computer Science, IEEE, pp 661–670
8. Chazelle BM, Lee DT (1986) On a circle placement problem. Computing 36(1-2):1–16
9. De Berg M, Van Kreveld M, Overmars M, Schwarzkopf O (1997) Computational geometry. In: Computational geometry, Springer, pp 1–17
10. Driemel A, Har-Peled S, Wenk C (2012) Approximating the Fréchet distance for realistic curves in near linear time. Discrete & Computational Geometry 48(1):94–127
11. Edelsbrunner H, Guibas LJ, Stolfi J (1986) Optimal point location in a monotone subdivision. SIAM Journal on Computing 15(2):317–340
12. Edelsbrunner H, Guibas L, Pach J, Pollack R, Seidel R, Sharir M (1992) Arrangements of curves in the planetopology, combinatorics, and algorithms. Theoretical Computer Science 92(2):319–336
13. Fogel E, Halperin D, Wein R (2012) CGAL arrangements and their applications: A step-by-step guide, vol 7. Springer Science & Business Media
14. Fort M, Sellarès JA, Valladares N (2014) Computing and visualizing popular places. Knowledge and information systems 40(2):411–437
15. Gilbert JR, Miller GL, Teng SH (1998) Geometric mesh partitioning: Implementation and experiments. SIAM Journal on Scientific Computing 19(6):2091–2110
16. Goodrich MT (1991) Intersecting line segments in parallel with an output-sensitive number of processors. SIAM Journal on Computing 20(4):737–755
17. Goodrich MT (1993) Constructing arrangements optimally in parallel. Discrete & Computational Geometry 9(4):371–385
18. Goodrich MT, Sitchinava N, Zhang Q (2011) Sorting, searching, and simulation in the mapreduce framework. arXiv preprint arXiv:11011902
19. Haran I (2006) Efficient point location in general planar subdivisions using landmarks. Tel Aviv University
20. Karloff H, Suri S, Vassilvitskii S (2010) A model of computation for mapreduce. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, SIAM, pp 938–948
21. Kaul A, Farouki RT (1995) Computing minkowski sums of plane curves. International Journal of Computational Geometry & Applications 5(04):413–432

22. Kedem K, Livne R, Pach J, Sharir M (1986) On the union of jordan regions and collision-free translational motion amidst polygonal obstacles. Discrete & Computational Geometry 1(1):59–71
23. Laube P, Imfeld S, Weibel R (2005) Discovering relative motion patterns in groups of moving point objects. International Journal of Geographical Information Science 19(6):639–668
24. Laube P, van Kreveld M, Imfeld S (2005) Finding remodetecting relative motion patterns in geospatial lifelines. In: Developments in spatial data handling, Springer, pp 201–215
25. Leighton FT (2014) Introduction to parallel algorithms and architectures: Arrays· trees· hypercubes. Elsevier
26. Oks E, Sharir M (2006) Minkowski sums of monotone and general simple polygons. Discrete & Computational Geometry 35(2):223–240
27. Overmars MH, Yap CK (1991) New upper bounds in klees measure problem. SIAM Journal on Computing 20(6):1034–1045
28. Pollack R, Sharir M, Sifrony S (1988) Separating two simple polygons by a sequence of translations. Discrete & Computational Geometry 3(2):123–136
29. Sarnak N, Tarjan RE (1986) Planar point location using persistent search trees. Communications of the ACM 29(7):669–679
30. Shakhnarovich G, Darrell T, Indyk P (2006) Theory, The MIT Press
31. Sharir M (1987) Efficient algorithms for planning purely translational collision-free motion in two and three dimensions. In: Proceedings. 1987 IEEE International Conference on Robotics and Automation, Citeseer, vol 4, pp 1326–1331
32. Toth CD, O'Rourke J, Goodman JE (2017) Handbook of discrete and computational geometry (Chapter 28). Chapman and Hall/CRC
33. Zheng K, Zheng Y, Yuan NJ, Shang S, Zhou X (2013) Online discovery of gathering patterns over trajectories. IEEE Transactions on Knowledge and Data Engineering 26(8):1974–1988
34. Zheng Y, Li Q, Chen Y, Xie X, Ma WY (2008) Understanding mobility based on gps data. In: Proceedings of the 10th international conference on Ubiquitous computing, pp 312–321
35. Zheng Y, Zhang L, Xie X, Ma WY (2009) Mining interesting locations and travel sequences from gps trajectories. In: Proceedings of the 18th international conference on World wide web, pp 791–800
36. Zheng Y, Xie X, Ma WY, et al. (2010) Geolife: A collaborative social networking service among user, location and trajectory. IEEE Data Eng Bull 33(2):32–39