The VLDB Journal (2009) 18:57–75
DOI 10.1007/s00778-007-0081-y

REGULAR PAPER

Approximate voronoi cell computation on spatial data streams

Mehdi Sharifzadeh · Cyrus Shahabi

Received: 18 August 2007 / Accepted: 20 August 2007 / Published online: 19 December 2007
© Springer-Verlag 2007

Abstract Several studies have exploited the properties of Voronoi diagrams to improve the efficiency of variations of the nearest neighbor search on stored datasets. However, the significance of Voronoi diagrams and their basic building blocks, Voronoi cells, has been neglected when the geometry data is incrementally becoming available as a data stream. In this paper, we study the problem of Voronoi cell computation for fixed 2-d site points when the locations of the neighboring sites arrive as a *spatial* data stream. We show that the non-streaming solution to the problem does not meet the memory requirements of many realistic scenarios over a sliding window. Hence, we propose AVC-SW, an approximate streaming algorithm that computes $(1+\varepsilon)$-approximations to the actual exact Voronoi cell in $O(\kappa)$ where κ is its sample size. With the sliding window model and random arrival of points, we show both analytically and experimentally that for given window size w and parameter k, AVC-SW reduces the expected memory requirements of the classic algorithm from $O(w)$ to $O(k\log(\frac{w}{k}+1))$ regardless of the distribution of the points in the 2-d space. This is a significant improvement for most of the real-world scenarios where $w \gg k$.

Keywords Voronoi cell · Spatial data stream · Sliding window · Approximation

1 Introduction

Different variations of the Voronoi diagrams have been theoretically studied in the field of computational geometry. The Voronoi diagram of a set of points, termed *site points*, partitions the space into a set of convex polygons so that each polygon contains exactly one site point of the set. The polygon corresponding to each point p covers the points in space that are closer to the point p than to any other site point. The computational geometry literature refers to the polygons as Voronoi cells, Dirichlet regions, Thiessen polytopes, or Voronoi polygons [4,19]. The database literature sometimes refers to the data points inside the cell of p as the *influence set* of the point p [15,23,24].

In many problems, instead of constructing the entire Voronoi diagram, it is sufficient to compute the Voronoi *cell* of a fixed point (or those of a small subset of all points) with respect to the set of its neighboring site points. The cell defines the set of data points/objects which are related to or assigned to the fixed point in the context of a problem. Korn and Muthukrishnan [15] describe four examples of the cell computation problem drawn from different spatial/vector space domains in which the influence set of a given point is required. Stanoi et al. [24] compute the Voronoi cell of a query point to retrieve its influence set from a database of points. Zhang et al. [25] determine the so-called *validity region* around a query point as the Voronoi cell of its nearest neighbor. The cell is the region within which the result of the nearest neighbor query remains valid as the location of the query point is changing. To provide an efficient similarity search mechanism in a peer-to-peer data network, Banaei-Kashani et al. [3] propose that each node maintains its Voronoi cell based on its local neighborhood in the content space. As a more abstract computational geometry problem, Arya and Vigneron [2] focus on approximating the exact Voronoi cell of a fixed point in order to build a simplified cell. The research goal in these studies is focused on improving the performance of the reverse nearest neighbor queries when the data points are stored as a massive dataset [15,24],

M. Sharifzadeh (✉) · C. Shahabi
Computer Science Department, University of Southern California,
Los Angeles, CA 90089-0781, USA
e-mail: sharifza@alumni.usc.edu

C. Shahabi
e-mail: shahabi@usc.edu

reducing the server-client communication overhead for a location-based service [25], providing an efficient access method [3], or simplifying an exact Voronoi cell [2]. They all assume that the entire set of neighboring points required to compute the cell is always available.

However, there are real-world scenarios in which the locations of the neighboring points become incrementally available through a *spatial* data stream. In this case, the contribution of any newly arrived point to the current Voronoi cell of the fixed point can easily be determined. Any point with no contribution to the cell is dropped. Hence, the space complexity of this approach is $O(|\upsilon(p)|)$ where $|\upsilon(p)|$ is the number of points contributing to the cell of point p. The more contributing points arrive, the smaller the Voronoi cell becomes. Therefore, the cell eventually becomes identical to the point itself which makes this update scheme unrealistic. Instead, real-world applications require that the contribution of any point to the cell end after some period of time (i.e., the point expires). In this case, we can no longer drop a newly arrived point which is not contributing to the cell. That is, to compute the exact Voronoi cell, we require to store *all* unexpired points arrived so far (i.e., $O(w)$ space complexity for w unexpired points). The reason is that even though an arriving point may not be changing the cell at the time of its arrival, it may cause a change in future due to the removal of old expired points. To be precise, assuming that any point expires after arriving w newer points, w arrivals and w expirations occur during the lifetime of a point. The expiration of any single point may cause *any* unexpired point to suddenly contribute to the cell while any new arrival may end a point's contribution (see the updates to the cell in Fig. 2). Therefore, no point can be dropped as even the points with no contribution at their arrival time might contribute later when other points' contributions end. This results in a large memory requirement per points in order to maintain its exact Voronoi cell. Therefore, in this paper, we study various techniques to keep only a subset of the arriving points and still maintain a "good enough" approximation of the point's Voronoi cell.

1.1 Motivation

As a motivating application, consider the sensor nodes in a sensor network deployed to continuously monitor a physical phenomena such as soil temperature. The average temperature of an area can be computed as the weighted average of the temperature values recorded by each sensor node to provide a seamless average, independent of the density of the network. In this case, each node's weight can be the area of its Voronoi cell [21]. Each immobile sensor node, knowing its fixed location, receives the locations of other mobile/immobile nodes as a spatial data stream and must independently build its Voronoi cell with respect to the recently received locations (e.g., those received in the last one hour). That is, each node requires to maintain its cell considering a *sliding window* over the stream of other nodes' locations. In Sect. 3, we show that a node *must* store *all* these locations for the duration of one hour (i.e., window size) to continuously keep its cell up-to-date as new points arrive and old points expire (e.g., the corresponding nodes stop operating).

As another example, consider the case where a combination of soldiers' wearable sensory devices and thousands of immobile sensor nodes deployed in a battlefield are used to guard an area. Each immobile node continuously receives the locations of the friendly soldiers and through its sensory devices detects the locations of the enemy soldiers. It maintains its Voronoi cell with respect to only the locations of friendly soldiers. Consequently, it can compute its reverse nearest neighbors as the enemy soldiers inside its Voronoi cell. As an autonomous nonhuman guard, the node directs its closest friendly soldier towards its location if the count of enemies is unusually large (i.e., its neighborhood is under a possible attack) or its Voronoi cell is becoming large (i.e., friendly soldiers are getting far from its neighborhood).

Our last example application is from the domain of simulation-based games.[1] Consider thousands of avatars simulating soldiers in a real battlefield. The behavior of each soldier is controlled by a process thread in the application. Each soldier (i.e., its representing thread) should decide which soldier of the opponent is the next one to shoot. The best candidates are those in its Voronoi cell with respect to the up-to-date locations of all other friendly soldiers. Here, when a soldier is shot, its location is expired. Considering this as a face-to-face fight, the movements of the soldiers can be ignored. Hence, the Voronoi cells of a set of *fixed* locations must be maintained where the locations of shot soldiers are continuously expiring.

Note that in real-world scenarios the number of unexpired points w might be very large. From an application perspective, assuming that any point expires *one day* after its arrival, a data stream with a rate of 40 points/min can result in 57,600 unexpired points at any time while a sensor node (e.g., Berkeley Mica2 Mote[2]) can only store up to 51,200 points of size 10 bytes each in its 512 Kb memory. This is assuming the unrealistic case that the entire memory space can be dedicated to this task! This shortage in memory introduces a need to compute an approximation of the Voronoi cell with deterministic error using only the available memory. On the other hand, many applications may require the computation of an approximate cell within a user's tolerable error. Consequently, an approximation algorithm is required to compute the cell with respect to the tolerable error utilizing the minimum memory space. Hence, one can consider two related optimization problems. Either minimize error for a

[1] http://www.ict.usc.edu/.

[2] http://www.xbow.com/.

fixed memory size or minimize memory requirement for a given error threshold. Our focus is on both of these optimization problems.

1.2 Contribution

In this paper, we study the general problem of computing the Voronoi cell of a fixed 2-d point p with respect to a *data stream* of site points. The point p could be the fixed location of an immobile sensor node in our motivating application.[3] To describe the preliminaries of our approach, we start with the simpler unrealistic case where no arrived point expires. That is, any single point arrived on the data stream so far may cause a change in the *current* Voronoi cell of p (this is called the *time series* model in [18]). Hence, at any point in time a node maintains its Voronoi cell with respect to the set of all the locations received so far. Next, we extend our approach to the more realistic scenario, the *sliding window* case, where the old points' effects on the cell terminate when they expire (and new points arrive). Here, the sensor node ignores the locations received earlier than a specific time when building its cell. Although a few research studies have focused recently to revisit classic computational geometry problems in a data stream framework (see [13] for a complete list), to the best of our knowledge no study has reconsidered the problem of building Voronoi cells in this framework.

Throughout the paper, our goal is to maintain an approximation to the exact Voronoi cell of a point p while the locations of the site points are arriving as a data stream. The core idea behind our approach is to maintain a minimum subset of site points including the closest ones to p in each direction and compute the Voronoi cell of p with respect to this subset instead of that of all the points. To reduce the $O(w)$ space complexity of the classic algorithm in the sliding window model where w is the window size, we first propose AVC, an approximation algorithm for the time series model which maintains only a sample of the streamed points using a similar sampling technique proposed for building radial histograms in [5] and for maintaining convex hull of data streams in [11]. AVC employs the classic algorithm to build the Voronoi cell of the sample as an *approximation* to the exact cell. The AVC's time complexity of an update per-point is $O(\kappa)$ where κ is the sample size. The value of κ is bounded by a single user-provided parameter k that is independent of the distribution of the points (i.e., $\kappa \leq k$).

Next, for the sliding window model, we propose an extension to AVC, AVC-SW, with the time complexity of $O(\log(\frac{w}{k}+1) + k\log(k))$. AVC-SW stores only the points which might become close to p in a future window. Now, our first optimization problem to minimize memory for a given approximation error is addressed as follows. First, we theoretically prove that one can determine the single parameter k of AVC (and AVC-SW) based on user's tolerable error ε (Sect. 6). Then, we show that both AVCs use this parameter to compute $(1+\varepsilon)$-approximations to the actual exact Voronoi cell. More precisely, we prove that if a point q is inside the approximate Voronoi cell of a point p, its distance to its closest site point is less than its distance to p by at most a factor of $1+\varepsilon$. Finally, for a uniform random arrival of points, we theoretically compute the expected sample size of AVC-SW (i.e., its required memory) in terms of the window size w and its single parameter k (Sect. 7). We show that the sample size is $O(k\log(\frac{w}{k}+1))$ regardless of the distribution of points in the 2-d space. It grows very slowly as the ratio of w over k increases. For instance, it is less than $20k$ for $w/k \leq 2.5 \times 10^8$. We similarly address the second optimization problem to minimize approximation error for a fixed memory size. In other words, given the window size w, we use Eq. 21 to compute the value of parameter k in terms of available memory κ. Then, we utilize Theorems 2 and 3 to determine the approximation error ε of AVC-SW's output.

In [22], we studied four different approaches to calculate weighted average of sensor measurements using each sensor's covered area as its weight. One of the approaches required the calculation of the Voronoi cell of each sensor, for which we employed our AVC-SW algorithm discussed in this paper. In this paper, however, we focus on the general problem of approximating Voronoi cells for different streaming scenarios. Here we propose two different streaming algorithms AVC and AVC-SW, investigate their solution spaces and set up the theoretical foundation to prove their optimality and guaranteed approximation errors. Hence, this paper exclusively contains the theoretical foundation of our streaming algorithms for approximating Voronoi cells in different data stream models.

2 Definitions

The Voronoi cell of a point p with respect to a given set of points $N \subset \Re^2$ is a *unique convex* polygon which includes all the points in the space $\Re^2$ that are closer to p than to the other points in the set N. Each edge of the polygon is a part of the perpendicular bisector line of the line segment connecting p to one of the points in the set. We call each of these edges a *Voronoi edge* and each of its end-points (vertices of the polygon) a *Voronoi vertex* of p. For each Voronoi edge of the point p, we refer to the corresponding point in N as a *Voronoi*

[3] Notice that mobile nodes also need to compute their Voronoi cells. However, the focus of this paper is on computing Voronoi cells for fixed points (e.g., the immobile nodes). Addressing the problem for moving points (e.g., mobile nodes) has its own challenges and is beyond the scope of this study. Notice, however, that in real-world applications immobile nodes are the only cheap and unattended nodes. Therefore, it is more likely that they suffer from memory limitation. Mobile nodes such as PDAs or laptops are mainly handled by human operators and have more memory spaces.

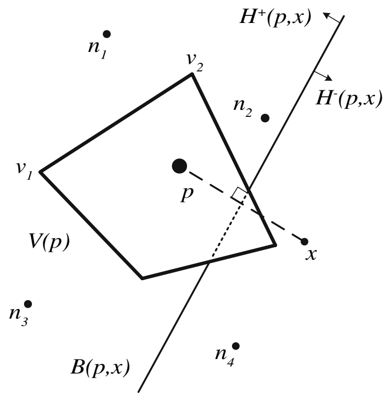

Fig. 1 The Voronoi cell of the point p with respect to $N = \{n_1, \ldots, n_4\}$. The bisector line corresponding to the new point x excludes all the points in $H^-(p, x)$ from $V(p)$

neighbor of p. Furthermore, the set N is usually called the set of *site* points. As an example, Fig. 1 shows the Voronoi cell of a point p as a quadrilateral generated given the set $N = \{n_1, \ldots, n_4\}$. The points n_1 and v_1, and the edge $\overline{v_1 v_2}$ are the corresponding Voronoi neighbor, vertex and edge of p, respectively. The definition of the Voronoi cell of a point in the 2-dimensional space $\Re^2$ follows:

Definition 1 If p is a 2-dimensional point, N is a set of n points in the 2-dimensional space $\Re^2$, and $D(., .)$ is Euclidean distance metric defined in the space, then $V(p)$, the Voronoi cell of the point p given set N, is defined as the unique convex polygon which contains all the points in the set $V_N(p)$:[4]

$$V_N(p) = \{q \in \Re^2 \mid \forall\, n \in N, D(q, p) \leq D(q, n)\}$$

We use $|pq|$, $\overline{pq}$, and $B(p, q)$ to denote the Euclidean distance between the points p and q, the line segment connecting them, and the perpendicular bisector line of this segment, respectively. We use $\upsilon(p)$ to refer to the set of Voronoi neighbors of p. It is clear that the Voronoi cell $V(p)$ can be computed using $\upsilon(p)$ in $|\upsilon(p)|$ steps and vice versa. Hence, we use $V(p)$ and $\upsilon(p)$ interchangeably throughout the paper.

3 The problem

Assume that we need to compute $V(p)$, the Voronoi cell of a given fixed point p with respect to a set of n site points N. For each point $q \in N$, the line $B(p, q)$ divides the space into two half-planes: $H^+(p, q)$ including p and $H^-(p, q)$ including q. For any point in $H^-(p, q)$, we say that $B(p, q)$ *excludes* the point from $V(p)$ (see Fig. 1). The trivial way to find $V(p)$ is to find the common intersection of all n half-planes $H^+(p, x)$ for all points $x \in N$. The edges of $V(p)$ form the boundary of the intersection. If we use linear programming, the intersection is computed in $O(n \log n)$ time and linear storage [4].

In this paper, we study two variations of the problem. First, consider the case when the points in N become incrementally available as a spatial data stream. That is, at each time instance t, we receive only one data tuple $\langle x, y\rangle$ representing the coordinates of a point $n_t \neq p$. Consequently, we update N to $N \cup \{n_t\}$. The point p could be the fixed location of an immobile sensor node in the motivating example of Sect. 1. The node incrementally receives the locations of the other nodes (i.e., points in N) through a data stream. Here, the update scheme of N is broadly referred to as *time series model* in the data stream literature [18]. Now the problem is to update $V(p)$ (or $\upsilon(p)$) according to the updates to N (i.e., when a new point x arrives). The classic solution is to find the intersection of $B(p, x)$ with Voronoi edges of p (i.e., edges of $V(p)$) and update $V(p)$ to the intersection of $V(p)$ and $H^+(p, x)$ (see Fig. 1).

The time complexity of each update is $O(|\upsilon(p)|)$ where $|\upsilon(p)|$ is the number of current Voronoi neighbors of p. This complexity is $O(|N|)$ for the worst case where any point in N is a Voronoi neighbor of p. Meanwhile, it is $O(1)$ on average as considering any point distribution the average number of vertices of a Voronoi cell is less than six [4]. Therefore, the approach meets the common requirement in data stream algorithms which dictates that the complexity of an update per-point must be sub-linear in time. Moreover, the space complexity of the solution is also $O(|\upsilon(p)|)$. The reason is that if $B(p, x)$ does not intersect with the current $V(p)$, we do not need to store the point x. Hence, we only store the Voronoi neighbors of p at each update time and drop the other points.

Even though the optimal algorithm to compute the exact Voronoi cell is cost effective, its complexity mainly depends on the distribution of the site points. The number of Voronoi neighbors of the point p depends on the position of the points in the stream. Therefore, the amount of space required to store the cell is not deterministic. This is critical in applications such as sensor networks with memory and power limitations.

Now consider the *sliding window* case when we are only interested in w most recent points. We discussed in Sect. 1 that this case represents real-world scenarios. Here, the goal is to maintain the Voronoi cell of a fixed point p with respect to the set of points arrived so far in a window W of fixed size.[5] With an example, we show that to compute the exact cell and

[4] We assume that such a bounded polygon exists. Futhermore, we assume that $p \notin N$. While this convention is different from the literature on Voronoi diagrams, the result is the same.

[5] While our motivating examples utilize time-based windows, we use fixed-sized windows throughout the paper. It is easy to specify a reasonable window size considering both the temporal size of a given time-based window and the application under study. Notice that the size of this window can be highly affected by the rate of updates on the data stream (e.g., bursty intervals require larger windows).

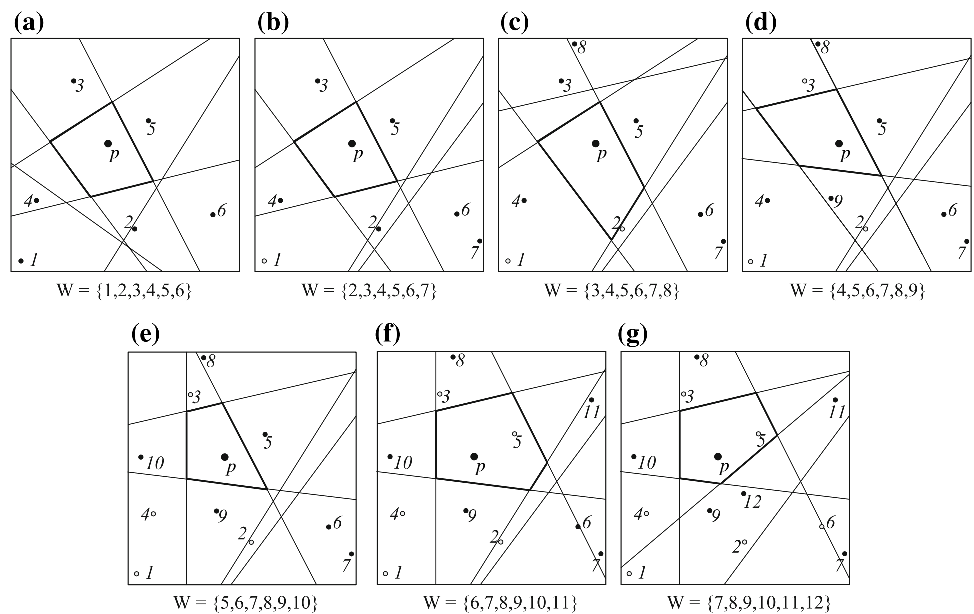

Fig. 2 The Voronoi cell of point p over a sliding window W of size 6 for seven subsequent time instances. The label of the each point shows its arrival order

keep it up-to-date at any point in time, we must store all unexpired points (i.e., those in the window). To illustrate, Fig. 2 shows the Voronoi cell of a point p over a window of size 6 for seven subsequent time instances. Each point is labeled by its arrival order (or time). The points shown as filled dots are within the current window (i.e., the set W) while empty dots show the others. Each figure snapshot shows only the bisector lines of the points in W. In Fig. 2a, when the point 6 arrives, its corresponding bisector does not intersect with the cell and hence it is not a Voronoi neighbor of p. However, later in Fig. 2c and f, the point 6 does become a Voronoi neighbor of p. On the other hand, the point 7 never becomes a Voronoi neighbor of p during any of the time instances when the point 7 is in the current window (Fig. 2b–g). This example shows that the classic algorithm cannot drop a new point (e.g., point 6) even though its corresponding bisector does not intersect currently with the cell. That is, the space complexity of the algorithm is $O(w)$ where w is the size of the window.[6] Therefore, the classic algorithm is too expensive in terms of memory requirements for realistic scenarios. Motivated by this observation, as a prerequisite we first propose an algorithm to maintain an approximate Voronoi cell in the general time series model (Sect. 4). In Sect. 7, we extend our algorithm to be applicable over a sliding window. Our main objective is to reduce the number of site points which are required to be stored for the average case.

[6] In fact, some of the old points could be dropped based on the location of the new point. However, the process of choosing these candidate points is significantly expensive for the classic algorithm (i.e., $O(w^2)$). Details are removed due to the lack of space.

4 The approximate voronoi cell algorithm (AVC)

In this section, we propose an approximation algorithm for the unrealistic time series case in order to extend it to the realistic sliding window case in Sect. 7. We want to maintain an approximation to the Voronoi cell of the point p while the site points in N are arriving as a data stream. The core idea behind our AVC algorithm is to maintain a minimum subset of N including the closest site points to p and compute the Voronoi cell of p with respect to this subset instead of N. This cell is an approximation to the exact Voronoi cell with respect to the *entire* N.

Figure 3 shows the pseudo-code of AVC. We divide the 2-d space using k vectors in k different directions. Each vector originates from the point p. Moreover, the angle between each pair of neighboring vectors is $\theta = 2\pi/k$. We will show in Sect. 6 that the value of k can be determined as a function of the user's tolerance for error. As Fig. 4a shows, the vectors partition the space into k identical *sectors*. For each sector S_i, we store a point $m(S_i)$, the closest site point to the point p which is inside S_i. We refer to this point as the *minimum point* of the sector S_i. The procedure **Init()** in Fig. 3 performs this initialization step. When a new point x arrives through the stream, first we find the sector S_x containing x. Then, we replace the minimum point of the sector ($m(S_x)$) with x if the point p is closer to the point x than to the point $m(S_x)$ (procedure **Update()**).

Now the Voronoi cell of p with respect to the set of k minimum points corresponding to k sectors ($M_N = \bigcup_{i=1}^{k}\{m(S_i)\}$) is an approximation of the actual Voronoi cell of p using all site points arrived so far (i.e., set N). It is clear that the approximate Voronoi cell of p, contains its actual Voronoi

	p : generator of $V(p)$
	k : parameter of AVC
	Procedure Init(point p, integer k)
1.	$\theta = 2\pi/k$;
2.	divide the 2-d space around p in to k sectors using k vectors originating from p
3.	for each sector S_i do
4.	point $m(S_i) = null$;
	x : newly arrived site point
	Procedure Update(point x)
1.	$S_x =$ sector containing x;
2.	$m = m(S_x)$;
3.	if $\|px\| < \|pm\|$ then
4.	$m(S_x) = x$;
	Function ApproximateVC(point p)
1.	set $M_N = \bigcup_{i=1}^{k} \{m(S_i)\}$;
2.	$V'(p) =$ Voronoi cell of p with respect to M_N;
3.	return $V'(p)$;

Fig. 3 Pseudo-code of AVC

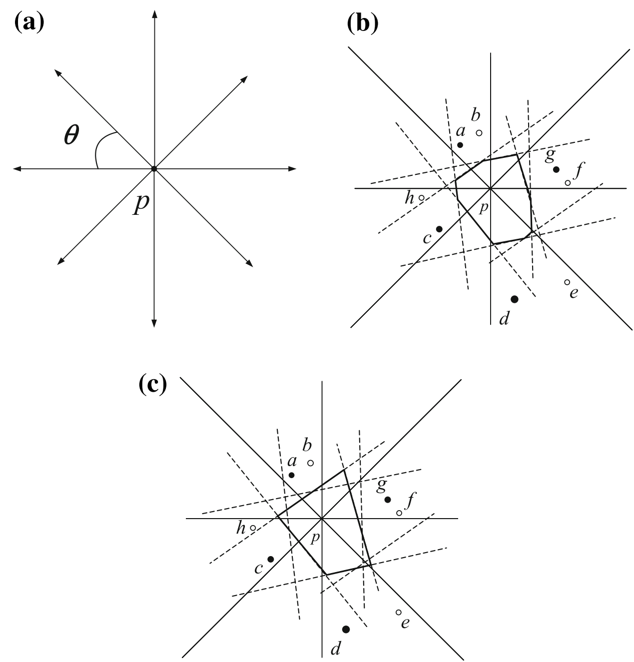

Fig. 4 **a** $k = 8$ vectors originating from p divide the space into k identical sectors, **b** the Voronoi cell of the point p, and **c** the approximate Voronoi cell of p

cell. We can compute the approximation in $O(k \log k)$ time and space at any time using the classic algorithm from the scratch. By incrementally updating this approximation on new point arrivals, the per-point computation can be reduced to $O(k)$. Furthermore, the time complexity of the per-point sample update is $O(1)$. Hence, the per-point update time of AVC including the time for updating both the sample and the approximation is $O(k)$. Therefore, AVC maintains a sample of size $\kappa = k$ to improve in terms of both time and space complexity over the classic algorithm specially when $k < |\upsilon(p)|$.

Throughout this paper, we use AVC(θ) to denote our algorithm with parameter θ in $k = 2\pi/\theta$ specifying the number of sectors. Furthermore, we use $V'(p)$ to refer to AVC's approximation to the Voronoi cell of the point p. Figure 4b shows the exact Voronoi cell of p with respect to the set $N = \{a, b, c, d, e, f, g, h\}$. Figure 4c shows $V'(p)$ created by AVC($\theta = \pi/4$). The filled dots in the figure are minimum points of the sectors while the empty dots are dropped by AVC.

5 Properties of AVC

In this section, we study different properties of the approximate Voronoi cell computed by the AVC algorithm. We use these properties to compute the approximation error of the algorithm in terms of the parameter θ.

A primary property of the Voronoi cell of a point p is that it contains none of the site points.[7] We intend to maintain this property for the approximate Voronoi cell. It is trivial that the output of AVC(θ) depends on both the value of θ and the distribution of the site points in N. However, we show that specific values of θ can be used in AVC to make some properties of its output independent from the distribution of the input points.

Lemma 1 *For any point p, $V'(p)$ computed by AVC(θ) does not contain any of the site points in N for any arbitrary set N if and only if θ is less than $\pi/3$.*

Proof See Appendix. □

Another property of the Voronoi cell of p is that the distance of any point inside (on) the cell to p is less than (equal to) its distance to any site point in N. We show that for any point inside (on) the approximate Voronoi cell of a point p, its distance to its closest site point in N is less than its distance to p by at most a small factor. We define the function $f(q)$ over the set of points q in the 2-d space as

$$f(q) = \frac{|qp|}{|qr|}, \tag{1}$$

where r is the closest site point to q in N. The property indicates that $f(q)$ is always less than or equal to 1 for the set of

[7] Note that $p \notin N$.

points inside or on the actual exact Voronoi cell $V(p)$ (note that $p \notin N$). Over this set, the function f reaches its upper bound (one) on the points on the boundary of $V(p)$. To study the approximation error of the AVC algorithm, we need to find the upper bound of the function over the set of points inside or on $V'(p)$. In particular, if a point is inside or on $V'(p)$ we find how far its distance to p from its distance to its actual closest point could be. Towards this end, we first locate the points where the maximum of $f(q)$ occurs.

Lemma 2 *Let q be a point inside or on the boundary of $V'(p)$, computed by AVC(θ) for a point p, and r be its closest site point in N. If $\theta < \pi/3$, the maximum of $f(q) = \frac{|qp|}{|qr|}$ over all points q occurs for a point on the boundary of $V'(p)$.*

Proof The proof is by contradiction. Assume that q with the maximum $f(q) = \frac{|qp|}{|qr|}$ is inside $V'(p)$ (not on its boundary). According to Lemma 1, as θ is less than $\pi/3$, the site point r is not inside $V'(p)$. Therefore, the line segment $\overline{qr}$ intersects with one of the edges of $V'(p)$ at a point a. First, we show that

$$\frac{|ap|}{|ar|} > \frac{|qp|}{|qr|}. \tag{2}$$

The points p, q, and r are either collinear or form a triangle. Figure 5a illustrates the first case. As q is between a and p, and a is between q and r, and all four points are on the same line, we have $|qp| < |ap|$ and $|qr| > |ar|$. Therefore, Eq. 2 holds. Figure 5b shows the second case illustrating the triangle $\triangle qpr$. In the figure, we have $\angle qpa = \alpha$, $\angle apr = \beta$, and $\angle qrp = \gamma$. In the triangle $\triangle qpr$, the law of sines yields

$$f(q) = \frac{|qp|}{|qr|} = \frac{\sin \gamma}{\sin(\alpha + \beta)}. \tag{3}$$

Meanwhile, using the same law in the triangle $\triangle apr$, we get

$$\frac{|ap|}{|ar|} = \frac{\sin \gamma}{\sin \beta}. \tag{4}$$

As r excludes q from $V(p)$, r is inside the circle $C(q)$ centered at q with a radius of $|qp|$. Therefore, in the triangle $\triangle qpr$ we have $\alpha + \beta < \pi/2$. Comparing $\alpha + \beta$ with β results in

$$\beta < \alpha + \beta < \frac{\pi}{2} \Rightarrow \sin \beta < \sin(\alpha + \beta). \tag{5}$$

Comparing Eqs. 3 and 4, and considering the inequality in Eq. 5 shows that Eq. 2 holds in the second case too. Let s be the closest point to a in N. So we have $|as| \leq |ar|$, therefore

$$f(a) = \frac{|ap|}{|as|} \geq \frac{|ap|}{|ar|} > \frac{|qp|}{|qr|} = f(q). \tag{6}$$

Equation 6 contradicts our assumption and shows that the point q with the maximum value for $f(q)$ must be on the boundary of $V'(p)$. □

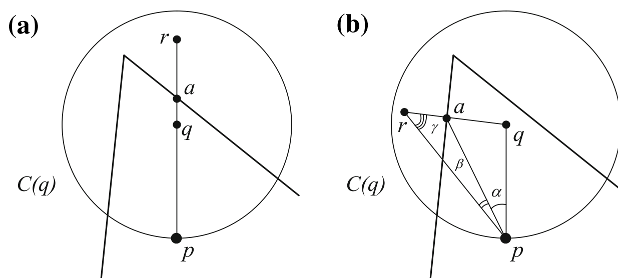

Fig. 5 The point q inside $V'(p)$ and its closest site point r where p, q, and r **a** are collinear, and **b** form the triangle $\triangle qpr$

6 Approximation error

We prove that the Voronoi cell computed by the AVC algorithm is a $(1 + \varepsilon)$-approximation to the actual Voronoi cell. More precisely, if a point q is inside the approximate Voronoi cell of a point p, its distance to its closest point in N is less than its distance to p by at most a factor of $1 + \varepsilon$ (i.e., $f(q) \leq 1 + \varepsilon$). We show that this difference is bounded and find the upper bound of ε for a given θ. Moreover, we prove that for a given ε, one can compute the largest θ for which AVC(θ) results in an approximation of tolerable error ε. To provide a proof, we first showed in Lemma 2 that the maximum of the function f occurs on the edges of the approximate Voronoi cell. In this section, for an arbitrary point q on the boundary of approximate Voronoi cell of p but outside its actual Voronoi cell, we consider the set of q's possible closest points in N which might have been dropped by the AVC algorithm. We find the maximum of $f(q)$ over the set of all points such as q.

Theorem 1 *If q is a point on the boundary of $V'(p)$ computed by the algorithm AVC(θ) and r is its closest site point in N, a certain positive ε can be found in terms of θ for which we have*

$$f(q) = \frac{|qp|}{|qr|} \leq 1 + \varepsilon. \tag{7}$$

Proof Let q be a point on one of the edges of the approximate Voronoi cell of p and outside the actual Voronoi cell of p (i.e., inside $H^-(p, r)$). That is, the closest point to q in $N \cup \{p\}$ is a point r other than p. The goal is to find the maximum of $|qp|/|qr|$. Towards this objective, we need to find the minimum of $|qr|$ as $|qp|$ is fixed for q. Hence, we locate the closest such a point r to q. It is clear that this point is not among k minimum points corresponding to the sectors (i.e., $r \notin M_N$). The reason is that if it was one of these points, the bisector line corresponding to $\overline{pr}$, $B(p, r)$, would have excluded q from the approximate Voronoi cell of p in the AVC algorithm. However, an exact Voronoi cell computation algorithm causes q to be outside $V(p)$. The locus of the points such as r whose corresponding bisector line,

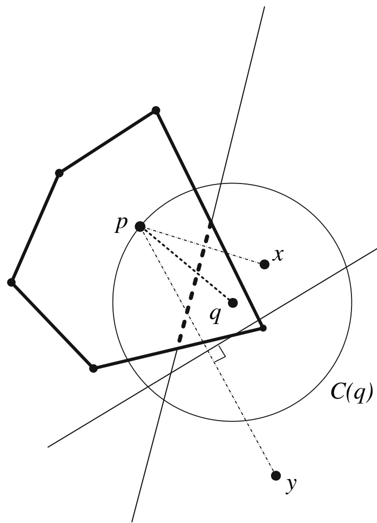

Fig. 6 The effect of x and y on the inclusion of q in the Voronoi cell of p

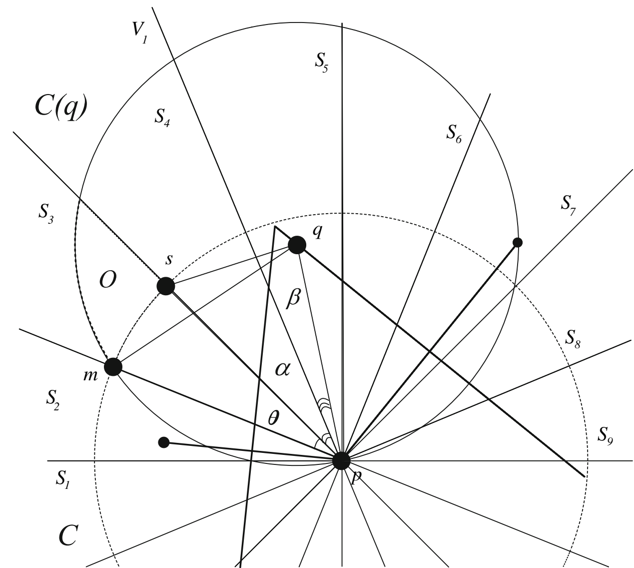

Fig. 7 The point q on one of the edges of $V'(p)$, the hidden point of the sector S_3, the angle β based on the location of q in S_5, and the angle α based on the location of S_3 and S_5

$B(p, r)$, excludes the point q from $V(p)$ is inside a circle centered at q with a radius of $|pq|$. We call this circle $C(q)$. To illustrate, consider the Voronoi cell of point p showed in Fig. 6. The figure shows a point q inside the Voronoi cell of p and the points x and y inside and outside the circle $C(q)$, respectively. The bisector line $B(p, x)$ intersects with the Voronoi cell causing q to be excluded from the cell. This is while, y which is outside the circle $C(q)$ has no effect on the inclusion of q in the cell.

Now consider all the sectors which intersect with the circle $C(q)$, namely $S_1, \ldots, S_9$ in Fig. 7. As q is on $V'(p)$ we can infer that for each of these sectors either it includes none of the points in N or its corresponding minimum point is outside the circle $C(q)$. However, because q is outside $V(p)$ the former case cannot be true for all of such sectors. That is, there must be at least one sector containing r with a minimum point outside the circle $C(q)$. This minimum point has caused the point r to be removed from our set of minimum points. Therefore, its corresponding bisector line has not excluded q from $V'(p)$ in the AVC algorithm. To find the minimum value of $|qr|$, we show how this happens and find the closest point r which has been removed because of this minimum point.

Consider one of the sectors that intersect with $C(q)$ (S_3 in Fig. 7). Let us locate the intersection point of the boundaries of the sector and the circumference of $C(q)$ which is closer to p (m in Fig. 7). The point m is the closest point to p inside S_3 and outside $C(q)$. Each of the points inside the sector S_3 and the circle $C(q)$ whose distance to p is more than $|mp|$ are removed by the AVC algorithm if m is the minimum point of S_3. The locus of these points is the intersection of (1) inside the sector S_3, (2) inside the circle $C(q)$, and (3) outside the circle centered at p with a radius of $|mp|$ (C). Figure 7 marks this intersection as O. The closest point to q in O is the point s. We refer to s as the *hidden* point of the sector S_3 with respect to q and $V'(p)$.

Now consider the hidden point of S_3. Let v_1 be the closest vector to q between $\overline{pq}$ and $\overline{ps}$. We use α and β to refer to the angles which $\overline{ps}$ and $\overline{pq}$ make with v_1, respectively (see Fig. 7). Notice that the location of q determines the value of β while α depends on the location of S_3 with reference to the sector containing q (i.e., S_5). Using the law of cosines in the triangle $\triangle pqs$ we have

$$|qs|^2 = |qp|^2 + |ps|^2 - 2|qp||ps|\cos(\alpha + \beta). \tag{8}$$

We have $|ps| = |pm|$ as both s and m reside on the circle C. Moreover, the triangle $\triangle pqm$ is an isosceles triangle as both points p and m are on the circle $C(q)$ ($|qm| = |qp|$). The angle between $\overline{pm}$ and $\overline{ps}$ is equal to θ as they are boundaries of the same sector. Now in the triangle $\triangle pqm$ we have $|pm| = 2|qp|\cos(\alpha + \beta + \theta)$. If we replace $|ps|$ with the value of $|pm|$ in Eq. 8, we get

$$\begin{aligned} |qs|^2 &= |qp|^2 + 4|qp|^2\cos^2(\alpha + \beta + \theta) \\ &\quad -4|qp|^2\cos(\alpha + \beta)\cos(\alpha + \beta + \theta). \end{aligned} \tag{9}$$

We define $F(\alpha, \beta, \theta)$ as

$$F(\alpha, \beta, \theta) = \sqrt{1 + 4\cos^2(\alpha + \beta + \theta) - 4\cos(\alpha + \beta)\cos(\alpha + \beta + \theta)} \tag{10}$$

and replace it in Eq. 9 to get

$$|qs| = |qp|F(\alpha + \beta + \theta). \tag{11}$$

For a given point q (with a fixed angle β) the value of $|qs|$ in the sector containing r is minimum over all other sectors

(with different values of α) that intersect with the circle $C(q)$. That is, as r is the closest point to q, the point q is closer to the hidden point of the sector containing r (i.e., s inside S_3 in Fig. 7) than to the hidden points of all of the sectors that intersect with the circle $C(q)$. To exploit the domain of the angle α, observe that as both points m and p are on the circle $C(q)$, the angle $\angle qpm = \alpha + \beta + \theta$ is not greater than $\pi/2$. Therefore we have

$$|qs| = \min_{\substack{0 \le \alpha \le \frac{\pi}{2} - \beta - \theta \\ \alpha = i\theta}} \left(|qp| F(\alpha + \beta + \theta) \right). \tag{12}$$

Notice that α is a multiple of θ as it is the angle between two vectors. So, we obtain

$$\frac{|qp|}{|qs|} = \frac{1}{\min_{\alpha = i\theta} F(\alpha + \beta + \theta)}. \tag{13}$$

Now as $|qr| \ge |qs|$ for the sector containing r, we have

$$f(q) = \frac{|qp|}{|qr|} \le \frac{|qp|}{|qs|}. \tag{14}$$

Therefore, the upper bound of $f(q)$ is not greater than the upper bound of $|qp|/|qs|$. The latter is the maximum of Eq. 13 over different points q (with different β angles). Hence, for any point such as q

$$f(q) \le \max_{0 \le \beta \le \theta} \left(\frac{1}{\min_{\alpha = i\theta} F(\alpha + \beta + \theta)} \right). \tag{15}$$

Equation 15 shows an upper bound for $f(q)$ as a function of α, β, and θ. In the remainder of the proof, we show that this function is bounded by another function in terms of θ. Let $\alpha_0 = \text{argmin}(|qs|)$ be the angle $\alpha = i\theta$ for which $|qs|$ is minimum. Figure 8a plots α_0/θ for different values of $k = 2\pi/\theta$ and β, and Fig. 8b shows a slice of this diagram for $k = 36$. As shown in both figures, the value of α_0 that determines the location of the point s depends on both θ and β. It means that if q (and β) changes in Fig. 7, the sector containing r might change to be S_4 (not S_3). In general, for a given θ the polar angle between the sector containing a point inside $V'(p)$ and its closest possible point which can be removed by AVC is not fixed.

Figure 8c plots the maximum of $f(q)$ for different values of θ. As the figure shows, $f(q)$ is less than 2 if k is greater than 8 (i.e., $\theta < \pi/4$). For any value of θ, one can extract a certain $\varepsilon = max(f(q)) - 1$ from Fig. 8c so that Eq. 7 holds. We observed that the maximum occurs when $\alpha + \beta + \theta \approx \pi/4$. To find a maximum function in terms of θ we plotted the following function in this diagram:

$$g(\theta) = \frac{1}{\sqrt{3 - 2\sqrt{2}\cos(\pi/4 - \theta)}}. \tag{16}$$

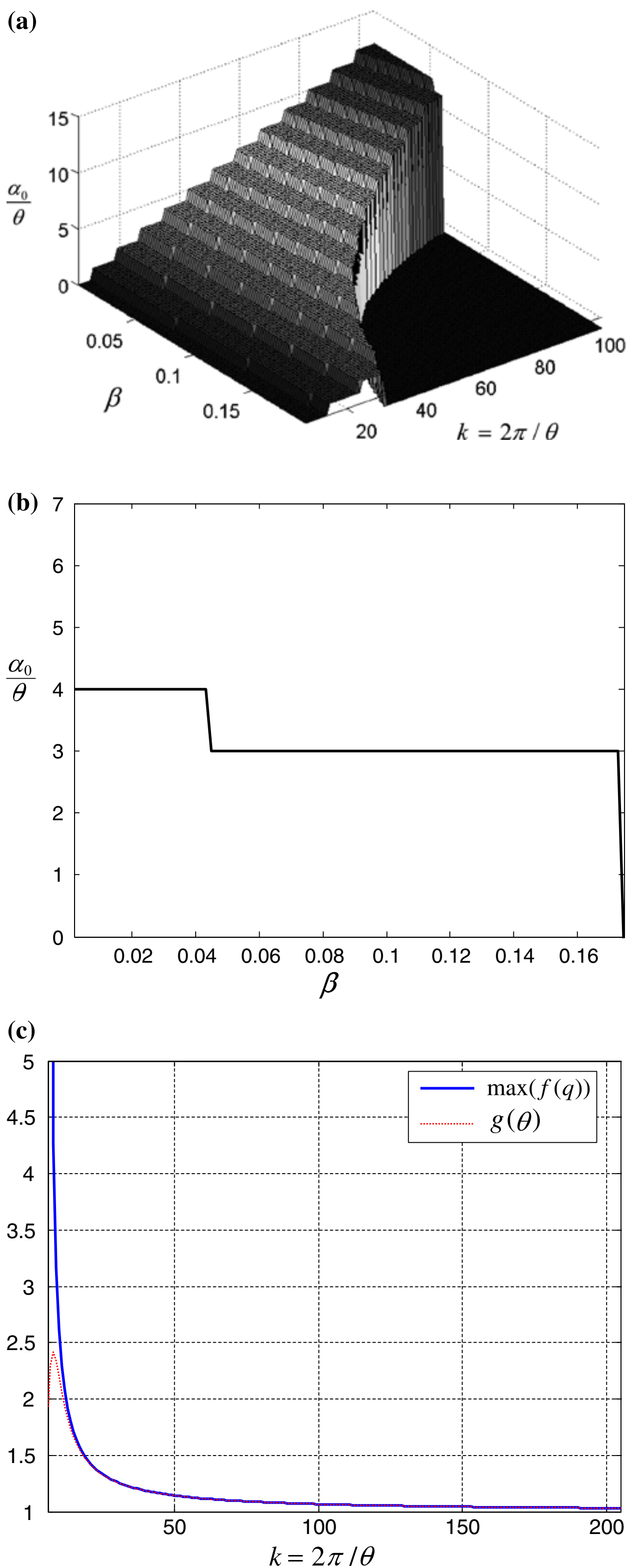

Fig. 8 **a** α_0/θ shows the location of the sector containing the closest hidden point to q for different values of β (in radians) and k , **b** α_0/θ for different values of β where $k = 36$ (i.e., $\theta = \pi/18$), **c** the upper bound of $f(q)$ for different values of k

Note that we reach $g(\theta)$ if we use $\alpha + \beta = \pi/4 - \theta$ in Eq. 13. The figure shows that for the values of $k > 8$, $g(\theta = 2\pi/k)$ is always greater than but very close to $f(q)$. Consequently, the upper bound of $f(q)$ is $g(\theta)$ if $k > 8$. It implies that Eq. 7 holds for $\varepsilon = g(\theta) - 1$. □

The immediate implication of the proof of Theorem 1 is that for a given ε, we can use $g(\theta)$ to compute the smallest k (i.e., the largest θ) to use with the AVC algorithm and result in an approximation of tolerable error ε.

Theorem 2 *For a given positive error bound ε, the largest θ, using which AVC(θ) can compute an approximate Voronoi cell with a maximum error of ε is computed from the equation $g(\theta) = 1 + \varepsilon$.*

We can also use Theorem 2 to compute the approximation error ε resulted from AVC(θ) given the value of parameter k. Moreover, our numerical exploration of the maximum of $f(q)$ for different values of k illustrated in Fig. 8c verifies the following theorem:

Theorem 3 *Given a reasonably large k (e.g., $k > 8$), AVC(θ) achieves an approximation of small error ε computed from the equation $\varepsilon = g(2\pi/k) - 1$.*

```
    p : generator of V(p)
    k : parameter of AVC-SW
    Procedure Init(point p, integer k)
1.    θ = 2π/k;
2.    divide the 2-d space around p in to k sectors
        using k vectors originating from p
3.    for each sector S_i do
4.        point m(S_i) = null;
5.        set M(S_i) = ∅;
-------------------------------------------------
    x : newly arrived site point
    Procedure Update(point x)
1.    S_e = sector containing expired point e;
2.    M(S_e) = M(S_e) − {e};
3.    m(S_e) = closest point to p in M(S_e);
4.    S_x = sector containing x;
5.    M(S_x) = M(S_x) ∪ {x};
6.    for each point y in S_x do
7.      if |py| > |px| then
8.          M(S_x) = M(S_x) − {y};
9.    m(S_x) = closest point to p in M(S_x);
```

Fig. 9 Pseudo-code of AVC-SW

7 AVC over sliding windows

In this section, we extend the AVC algorithm to be applicable to the sliding window model. With this model, the goal is to maintain the Voronoi cell of a point p with respect to the set of w recent points. With a window of fixed size w, when the new point n_t is arrived through the data stream, we update the set of site points N to exclude its oldest point and include n_t. We say the oldest point has *expired*.

In Sect. 3, we showed that a classic Voronoi cell computation algorithm cannot be used over a sliding window. The algorithm is not scalable to data stream rate and window size as it costs $O(w)$ memory. Likewise, the AVC algorithm is prone to the same problems. As an example, assume that AVC stores the minimum point m corresponding to the sector S at time t. Assume that according to the window size, m expires at time $t' > t$ (i.e., when we receive w points after t). Consider the set of site points that arrive during the time range $(t, t']$ and reside in the sector S. If p is closer to m than to any of these points, AVC drops all of them and maintains m as the minimum point of S. However, as soon as m expires, the minimum point of S needs to be updated to the point m', the closest point to p in S. Hence, AVC must store m'. Applying the reasoning recursively renders that AVC must store all the points in the window. In the remainder of this section, we extend the AVC algorithm to overcome this shortcoming by storing only the points which might become a minimum point in a future window.

With the AVC algorithm, we require to maintain the minimum points of each sector for the current window. The main idea behind our extension (AVC-SW) is that for each sector, we store any point which might be the minimum point in a future window before it expires. Therefore, we store all points upon their arrival. However, if p is closer to the newly arrived point than to any point x, previously stored for the corresponding sector, we drop x.

7.1 The AVC-SW algorithm

Let w be the window size, and at each time instance t, only one point arrives. We employ the same vectors of AVC to divide the space into sectors (see Sect. 4). For each sector S_i, we store the minimum point $m(S_i)$ and a set of points $M(S_i)$. This set includes all the points which can possibly be minimum points in the future windows. We initialize $m(S_i)$ and $M(S_i)$ to null for all sectors S_i before we start processing the data stream. Figure 9 shows the pseudo-code of the initialization and update steps of AVC-SW.

For each new point x, first we find the expired point among members of all $M(S_i)$ sets and delete it. Second, we find the sector S_x containing x and add x to $M(S_x)$. Third, we delete any point y in $M(S_x)$ if $|py| > |px|$. These points will never become minimum point of their sector in a future window.

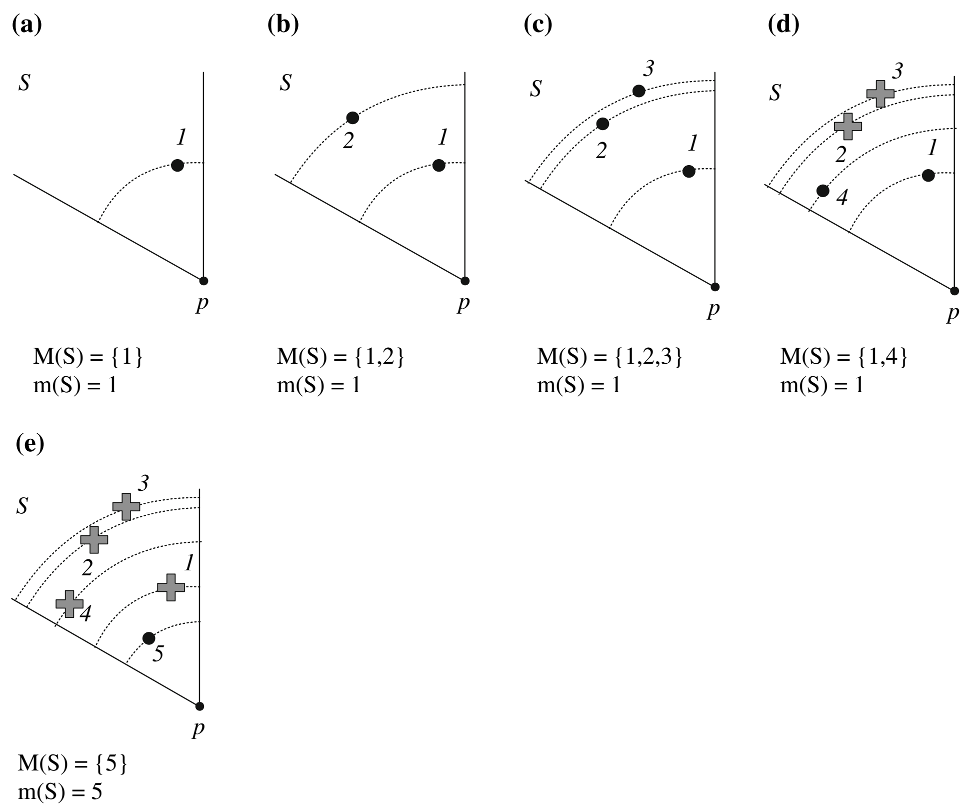

Fig. 10 AVC-SW updates the set $M(S)$ and the minimum point $m(S)$ for each of the 5 arriving points in the sector S. Assume that none of these points expire during this illustration

Finally, we set $m(S_x)$ to the closest point to p in $M(S_x)$ (see procedure **Update()** in Fig. 9). Similar to AVC, the Voronoi cell of p with respect to the set of k minimum points ($m(S_i)$) corresponding to k sectors is the approximation of the actual Voronoi cell of p. That is, AVC-SW employs the function **ApproximateVC()** shown in Fig. 3. The properties of $V'(p)$ and the approximation error analysis discussed in Sects. 5 and 6 also hold for the output of AVC-SW. The sample size of AVC-SW is computed as $\kappa = \sum_{i=1}^{k} |M(S_i)|$, where $|M(S_i)|$ is the cardinality of the set $M(S_i)$.

Figure 10 illustrates how AVC-SW maintains the minimum points of the sector S. The figure shows only the times when the new point is inside the sector S. Assume that none of these points expires during these time instances. As shown in Fig. 10a, the point 1 is the current minimum point of S. When the points 2 and 3 arrive in Fig. 10b and c, respectively, AVC-SW adds them to $M(S)$ as they might become the minimum point of S when 1 expires. However, 1 is still the current minimum point of S in the window. In Fig. 10d, the point 4 arrives. As 4 is in all future windows in which 2 or 3 exist and p is closer to 4 than to 2 and 3, we delete 2 and 3. Finally, the point 5 arrives in Fig. 10e and causes the update to the minimum point and deletion of all the points in $M(S)$.

In general case, the space requirements of AVC-SW is less than $O(w)$ as we drop the portion of the points that are unlikely to be a minimum point. However, in the worst case, when the points of each sector arrive in the increasing order of their distance to p, AVC-SW stores all of them (see Fig10a–c). In Sect. 7.2, we study the average sample size of AVC-SW.

7.2 Space complexity analysis

In this section, we theoretically find an upper bound for the expected number of points stored by AVC-SW where the points are arriving in a uniform random order. Notice that we make *no* assumption on the distribution of the points in the 2-d space. To study the variance of the sample size of AVC-SW, we also find the probability that the algorithm stores x points out of n unexpired points of each sector.

Given the window size w and the number of sectors k, assume that each sector S_i includes n_i points of the window (i.e., unexpired points). Hence, we have $\sum_{i=1}^{k} n_i = w$. Assume that AVC-SW stores $A(S_i)$ points of n_i points contained in sector S_i. That is, $A(S_i)$ is the cardinality of the set $M(S_i)$ for sector S_i. Hence, the total number of points stored by AVC-SW will be $\kappa = \sum_{i=1}^{k} A(S_i)$. We first show that the expected value of $A(S_i)$ is $\Theta(\log n_i)$ if $n_i > 1$.

Lemma 3 *Out of $n > 1$ unexpired points which arrive in sector S in a uniform random order, the expected number of points stored by AVC-SW is $\Theta(\log n)$.*

Proof AVC-SW uses the order of the points received in each sector and their distances to the center point p to decide whether they must be stored or dropped. For a sector S,

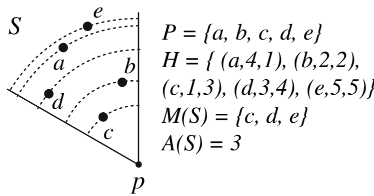

Fig. 11 Space analysis of AVC-SW

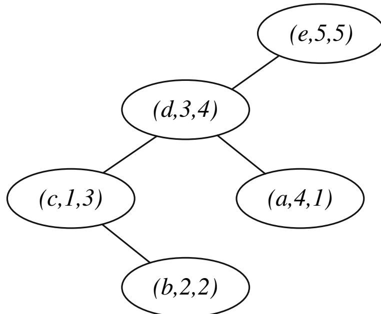

Fig. 12 The treap built on set H shown in Fig. 11

assume that $P = \{p_1, p_2, \ldots, p_n\}$ is the set of n points of the current window which are inside S. Now consider the set H which contains one 3-tuple item (p_i, r_i, t_i) for each $p_i \in P$. Integer $1 \le r_i \le n$ is the rank of the point p_i if we sort points in P based on their increasing distance to the point p and rank them accordingly.[8] Similarly, integer $1 \le t_i \le n$ is the rank of the point p_i if we sort points in P based on their arrival time and then rank them. To illustrate, Fig. 11 shows the sector S including five points a, b, c, d, and e, all in the current window. They have arrived in alphabetical order with a and e as the oldest and the most recent points, respectively. AVC-SW stores points c, d, and e so $A(S)$ is equal to three.

Now consider the set H. There is a unique treap [17] on members of set H considering r_i and t_i as the *key* and *priority* values of (p_i, r_i, t_i), respectively. Treap is a binary tree in which each node contains two values: a key and a priority. The keys satisfy the BST property while the priority values satisfy the heap property. To illustrate, Fig. 12 shows the unique treap built on the set H corresponding to the points shown in the example of Fig. 11. It is not hard to realize that $M(S)$ includes only the points on the left spine of this treap. As shown in the figure, the left spine of the treap corresponds to all the points of P that AVC-SW stores (i.e., points c, d, and e). Therefore, $A(S)$ is the length of the left spine of the treap.

Here, we intend to compute the expected value of $A(S)$. Both n-tuples $R = (r_1, \ldots, r_n)$ and $T = (t_1, \ldots, t_n)$ are clearly random permutations of positive integers less than or equal n. Depending on the distances of points in set P to the center point p, R could be any of $n!$ permutations of these numbers. Similarly, T is one of these $n!$ permutations depending on the arrival time of points in P. Therefore, the expected value of $A(S)$ is the expected length of the left spine of the treap built on the set H with random distinct key-priority pair values r_i and t_i.

It is known that the expected length of the left/right spine of a random treap of n nodes is the nth harmonic number H_n [17]. Therefore, assuming a random arrival of points inside sector S, the expected value of $A(S)$ is equal to H_n. That is, AVC-SW on average stores only H_n points out of n unexpired points inside sector S.

The harmonic number H_n has proven to be increasing slowly as n increases. For $n \le 2.5 \times 10^8$, it is still less than 20. Furthermore, to get H_n greater than 100, n must be greater than 1.509×10^{43} [7]. Seidel and Aragon [20] prove that H_n, the expected length of left/right spine of a treap of n distinct values, is $\Theta(\log n)$. Therefore, AVC-SW's expected sample size for $n > 1$ points in sector S is $\Theta(\log n)$. □

It is clear that if the sector S_i includes no unexpired point then AVC-SW stores no point corresponding to S_i (i.e., $n_i = 0 \Rightarrow A(S_i) = 0$). Moreover, if S_i includes only one unexpired point then AVC-SW stores this point (i.e., $n_i = 1 \Rightarrow A(S_i) = 1$). Now, as we have $\log(n+1) \simeq \log n$ for large values of n, we provide the following to generalize the result of Lemma 3 for all values of n:

Lemma 4 *Out of n unexpired points which arrive in sector S in a random order, the expected number of points stored by AVC-SW is $\Theta(\log(n+1))$.*

Now we find an upper bound for the expected number of points stored by AVC-SW for an arbitrary distribution of points in the space. We show that the sample size κ reaches this bound when all sectors S_i include the same number of points (i.e., $n_i = w/k$).

Theorem 4 *Given the window size w and the number of sectors k, the expected sample size of AVC-SW is $O(k \log (\frac{w}{k} + 1))$.*

Proof With k sectors S_i, AVC-SW's sample size κ is $\sum_{i=1}^{k} A(S_i)$. According to Lemma 4, the expected value of $A(S_i)$ is $\Theta(\log(n_i + 1))$. Therefore, we have

$$\mathrm{E}(\kappa) = c \sum_{i=1}^{k} \log(n_i + 1), \tag{17}$$

where c is a constant considering the definition of Θ notation. Pushing the log function outside the sum, we get

$$\mathrm{E}(\kappa) = c \log \prod_{i=1}^{k} (n_i + 1). \tag{18}$$

[8] To break the ties, for two points with the same distance to p, we insert the point with greater polar angle with the x-axis after the one with the smaller angle.

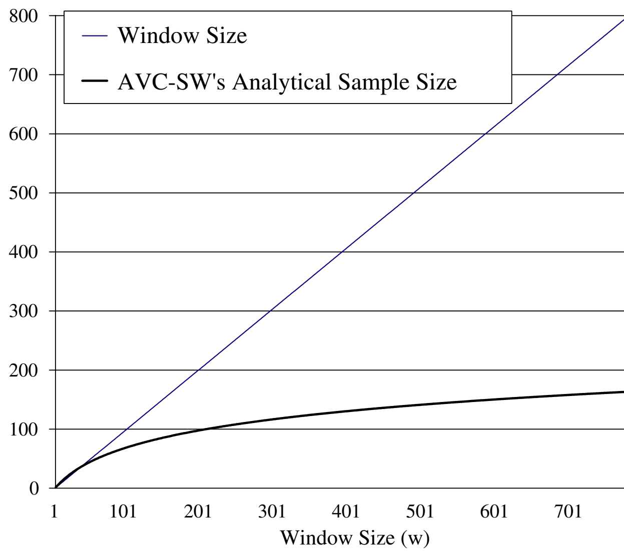

Fig. 13 Analytical expected number of points stored by AVC-SW (i.e., AVC-SW's sample size κ) for different values of w when $k = 36$

It is well known that the arithmetic mean of a set of numbers is an upper bound for their geometric mean. That is, for the set of $(n_i + 1)$'s we have

$$\prod_{i=1}^{k}(n_i + 1)^{\frac{1}{k}} \leq \frac{1}{k}\sum_{i=1}^{k}(n_i + 1). \tag{19}$$

Now, we know that the sum of all n_i's is fixed as they are all the points inside the current window of size w (i.e., $\sum_{i=1}^{k} n_i = w$). Therefore, we have $\sum_{i=1}^{k}(n_i + 1) = w + k$ and hence the following holds:

$$\prod_{i=1}^{k}(n_i + 1) \leq \frac{(w + k)^k}{k^k}. \tag{20}$$

Replacing the product of $(n_i + 1)$'s in the left side of Eq. 18 with the right side of Eq. 20, we get

$$\mathrm{E}(\kappa) \leq ck \log\left(\frac{w}{k} + 1\right). \tag{21}$$

Hence, we conclude that the expected sample size of AVC-SW is $O(k \log(\frac{w}{k} + 1))$. □

The result proved in Theorem 4 is independent from the distribution of points in the 2-d space. The expected value $\mathrm{E}(\kappa)$ is less than $20k$ when $w \leq 2.5 \times 10^8 k$. Figure 13 shows the function $k \log(\frac{w}{k} + 1)$ for $k = 36$ varying the window size.

So far, we theoretically showed that the expected sample size of AVC-SW is significantly less than the window size. To show that even for the general case AVC-SW's sample size is small with a high probability, we find the variance of the number of unexpired points inside each sector S which are stored by AVC-SW.

Theorem 5 *Let $\mathbf{P}_n(x)$ be the probability that AVC-SW stores x points out of n unexpired points which arrive in sector S in a uniform random order. We have the following*:

$$\mathbf{P}_n(x) = \begin{cases} 1/n & \text{if } x = 1 \\ (\mathbf{P}_{n-1}(x-1)+ & \\ (n-1)\mathbf{P}_{n-1}(x))/n & \text{if } 1 < x < n \\ 1/n! & \text{if } x = n \end{cases} \tag{22}$$

Proof We assume that the sector S includes n unexpired points $P = \{p_1, \ldots, p_n\}$. We also assume that the point p_i is the i-th arrived point which is inside S. The n-tuple $R = (r_1, \ldots, r_n)$ includes the rank r_i of p_i if we sort the points in P based on their increasing distance to the point p (the generator of $V(p)$) and rank them accordingly. For example in Fig. 11, we have $R = (4, 2, 1, 3, 5)$. It is clear that AVC-SW only uses the ranks in R to select the points which must be stored. If the points arrive in S in a uniform random order, then with an equal probability R is one of $n!$ permutations of positive integers less than $n + 1$. Assume that T_n is the set of all of these permutations. Let $N(n, x)$ be the number of permutations in T_n for which AVC-SW stores $x \leq n$ points. Therefore, $\mathbf{P}_n(x)$, the probability that AVC-SW stores x points out of n unexpired points of sector S is

$$\mathbf{P}_n(x) = \frac{N(n, x)}{n!}. \tag{23}$$

To compute the probability $\mathbf{P}_n(x)$, we first try to find $N(n, x)$. It is clear that $N(1, 1) = 1$. For $n > 1$, assume that out of n points in the set P corresponding to the permutation $R \in T_n$, AVC-SW stores only $\mathbb{N}(R) = x$ points. Consider the following three cases:

Case 1) $x = 1$: This case happens only when we have $R = R^1 = (r_1, \ldots, r_{n-1}, 1)$. That is, the newest point p_n arrived in sector S is the closest point to the generator p in the set P. Hence, upon arrival of p_n AVC-SW drops all other points and stores only p_n. The number of permutations similar to R^1 is $(n - 1)!$ as the last item of R^1 is fixed ($r_n = 1$) and the other $n - 1$ items can be selected from integers in the range $[2, n]$. Therefore, we have $N(n, 1) = (n - 1)!$.

Case 2) $x = n$: This case happens only when we have $R = R^2 = (1, 2, \ldots, n)$. That is, AVC-SW receives the points in P in the increasing order of their distance to the generator p. As we mentioned in Sect. 7.1, this is the worst case where AVC-SW stores all the n points of the sector. R^2 is unique among all permutations in T_n and hence we have $N(n, n) = 1$.

Case 3) $1 < x < n$: This case includes all other permutations $R \in T_n$ which does not match with those described as the previous two cases. Here, we find $N(n, x)$ recursively in terms of $N(n - 1, x - 1)$ and $N(n - 1, x)$. Let R' be the permutation of $n - 1$ integers resulted after removing n from R. Depending on the order of items in R, one of the two following subcases can happen:

Case 3.1) $R = (r_1, \ldots, r_{n-1}, n)$: The last item in R is equal to n and hence we have $R' = (r_1, \ldots, r_{n-1})$. For instance, in Fig. 11 where $n = 5$, we have $R = (4, 2, 1, 3, 5)$ and $R' = (4, 2, 1, 3)$. Obviously, AVC-SW stores the newest point p_n ranked as n as it is the farthest point from p in sector S. By assumption we have $\mathbb{N}(R) = x$ and hence AVC-SW must have stored $x - 1$ points from the other $n - 1$ points ranked as R'. According to our definition, there are $N(n-1, x-1)$ permutations such as $R' = (r_1, \ldots, r_{n-1})$ in T_{n-1} for each there is a single permutation $R = (r_1, \ldots, r_{n-1}, n)$ in T_n.

Case 3.2) $R = (r_1, \ldots, n, \ldots, r_n)$: An intermediate item in R is equal to n (i.e., $r_i = n$ for some $i \neq n$). For example for $n = 5$ and $R = (2, 3, 5, 1, 4)$ we get $R' = (2, 3, 1, 4)$. In this case, the point p_i with the rank $r_i = n$ is not stored by AVC-SW as there is at least one newer point p_{i+1} which is closer to the generator p (i.e., $r_{i+1} < r_i = n$). As we have $\mathbb{N}(R) = x$, AVC-SW must have stored x points from the other $n - 1$ points ranked as R'. Again, by definition there are $N(n - 1, x)$ permutations such as R' in T_{n-1}. However, as n could be any of the $n - 1$ intermediate items in R, for each R' in T_{n-1} there are $n - 1$ corresponding permutations R in T_n. For example, $R' = (2, 3, 1, 4)$ can be resulted by removing 5 from (5, 2, 3, 1, 4), (2, 5, 3, 1, 4), (2, 3, 5, 1, 4) and (2, 3, 1, 5, 4).

Considering both subcases of case 3, we get $N(n, x) = N(n - 1, x - 1) + (n - 1)N(n - 1, x)$ when $1 < x < n$. Equation 24 summarizes the final result for all values of x described through the above three cases:

$$N(n, x) = \begin{cases} (n-1)! & \text{if } x = 1 \\ N(n-1, x-1) + (n-1)N(n-1, x) & \text{if } 1 < x < n \\ 1 & \text{if } x = n. \end{cases} \tag{24}$$

Now we use Eq. 23 and divide both sides of Eq. 24 to $n!$ to get $\mathbf{P}_n(x)$:

$$\mathbf{P}_n(x) = \begin{cases} 1/n & \text{if } x = 1 \\ N(n-1, x-1)/n! + (n-1)N(n-1, x)/n! & \text{if } 1 < x < n \\ 1/n! & \text{if } x = n. \end{cases} \tag{25}$$

Finally, we replace $N(n-1, x-1) = (n-1)!\,\mathbf{P}_{n-1}(x-1)$ and $N(n-1, x) = (n-1)!\,\mathbf{P}_{n-1}(x)$ in Eq. 25 to get Eq. 22. □

Figure 14 illustrates $\mathbf{P}_{36}(x)$ for different values of x. It shows the probability that AVC-SW store x points out of 36 unexpired points which are inside a sector. As the figure shows, the distribution is skewed significantly towards the left (smaller values of x). The figure depicts that the probability that AVC-SW store even 12 points out of 36 unexpired points is 0.00005. Also, the figure shows that most likely the number of points stored by AVC-SW is $\lfloor \log 36 \rfloor = 5$ points (0.25 as the highest probability $\mathbf{P}_{36}(5)$).

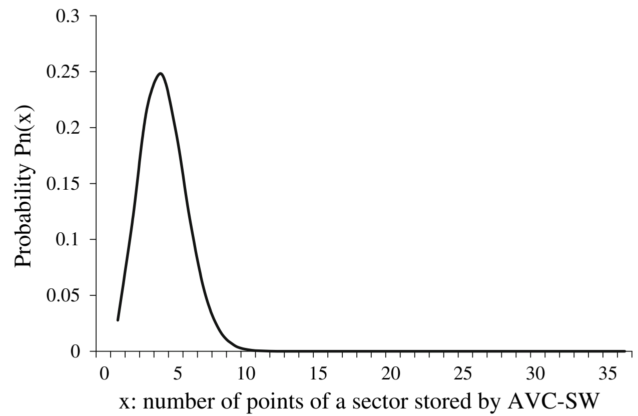

Fig. 14 $\mathbf{P}_n(x)$: Probability distribution of the number of points stored by AVC-SW out of $n = 36$ points in a sector

7.3 Time complexity

Theorem 4 also shows that the expected number of points stored for each sector is $O(\log(\frac{w}{k} + 1))$. Per each update, AVC-SW requires to remove the expired point from and add the new point to their corresponding sectors and also update the minimum point of these sectors. This step takes $O(\log(\frac{w}{k} + 1))$ time. The algorithm also builds $V(p)$ using the k minimum points from scratch which takes $O(k \log(k))$ time. Hence, the per-point update time of AVC-SW is $O(\log(\frac{w}{k} + 1) + k \log(k))$.

8 Performance evaluation

We conducted extensive experiments with both synthetic and real-world datasets to evaluate the average space used by AVC-SW and its approximation error for different values of parameter k and window size w. The experiments were all performed on a DELL Precision 470 with Xeon 3.2 GHz processor and 3 GB of RAM.

In the first set of experiments, we synthetically generated data streams of 1,000 points uniformly distributed inside a circle. These points are randomly chosen to arrive as a spatial data stream. We used the center of the circle as the fixed point p, applied AVC-SW's sampling algorithm on the stream and computed the average number of stored points during 100 different runs. Figure 15a illustrates the average sample size (κ) of AVC-SW for three different window sizes when we vary the parameter k. It shows that when k is less than the window size (e.g., k = 67 and w = 400), the sample size is far less than w. Figure 15b shows the same measurement for $k = 36$ and different window sizes. It shows up to 80% reduction in memory requirement for $k = 36$ and large windows.

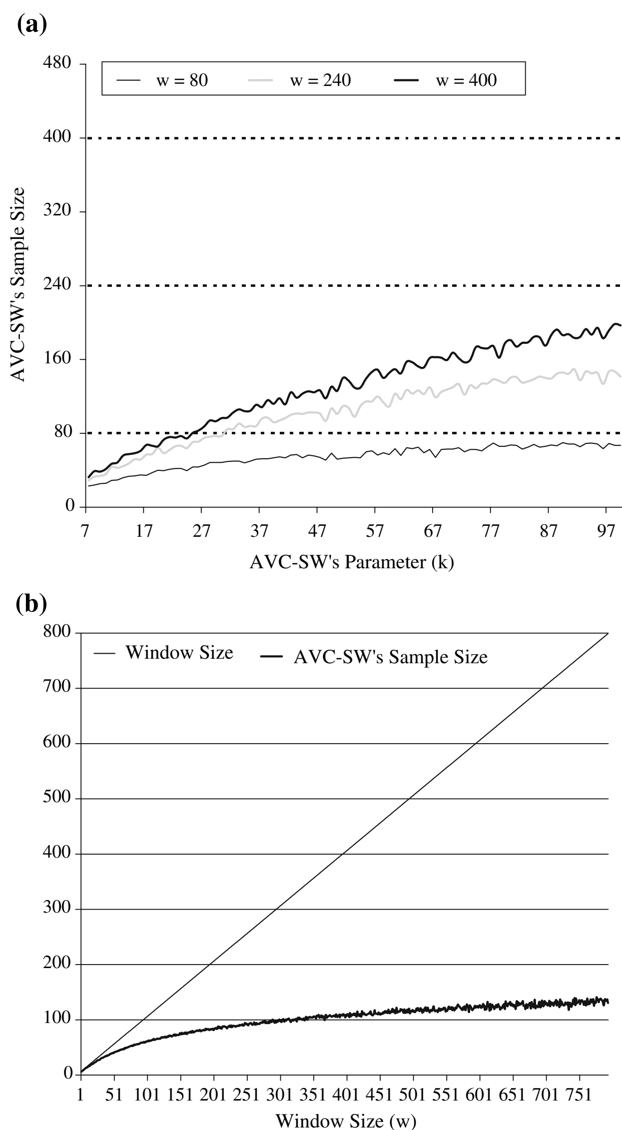

Fig. 15 Synthetic dataset: average number of points stored by AVC-SW (i.e., AVC-SW's sample size) for **a** $w = 80$, 240, and 400 and different values of k, and **b** different values of w when $k = 36$

Notice that the number of sectors k in Fig. 15b is the same as that of results shown in Fig. 13. Comparing the two figures shows that the analytical expected sample size computed in Eq. 21 completely supports the sample size computed by our experiments.

In the second set of experiments, we used three real-world datasets to generate our spatial data streams:

1. **USGS** dataset was obtained from the U.S. Geological Survey (USGS).[9] The dataset consists of 950, 000 locations of different businesses (e.g., churches) in the entire US.
2. **NE** dataset contains 123, 593 locations in New York, Philadelphia and Boston. Hence, the locations are clustered into three clusters and represent almost uniformly distributed rural areas and smaller population centers.[10]
3. **GRC** dataset includes the locations of 5,922 cities and villages in Greece.[10]

During each of 100 runs for each dataset, we chose the locations randomly to arrive as a spatial data stream. Then, we used AVC-SW to approximate the Voronoi cell of a point randomly chosen inside the minimum bounding box of the dataset.[11] Subsequently, we measured the average number of points stored by AVC-SW. We also measured the maximum error $\bar{\varepsilon}$ of the approximate Voronoi cell for each of window snapshots. This error $\bar{\varepsilon}$ is the maximum value of $f(q) - 1$ that occurs for a point q on the boundary of the Voronoi cell (Lemma 2).

Figure 16a shows the average sample size of AVC-SW using the **USGS** dataset for window sizes 80, 240, and 800 when parameter k increases from 7 to 100. Comparing this result with the corresponding result illustrated in Fig. 15a shows that AVC-SW requires noticeably less space for the real-world dataset. That is, it achieves better performance with the point distribution of the real **USGS** dataset. For example, when k is 70, AVC-SW stores only 38 points out of 80 unexpired **USGS** points while it stores 63 points of the uniformly distributed synthetic dataset.

Figure 16b illustrates the average sample size of AVC-SW for sliding windows of different sizes over all three real datasets when $k = 36$. The figure shows that AVC-SW stores only 10% of a window of 700 **USGS** points. This is almost half of the size of the sample which is stored by AVC-SW for a window of the same size over the synthetic dataset. We stated in Sect. 7.2 that with a random arrival of points, the expected sample size of AVC-SW reaches its maximum value when the points are equally distributed among the sectors. Now, the intuition here is that the non-uniform distribution of the points of the **USGS** dataset causes that the sectors containing centers of skewed areas get much more points than others. As a result, the sample size of AVC-SW is much less than its upper bound computed in Theorem 4. The figure shows that AVC-SW stores more points for the sliding windows of the same size over **GRC** and **NE** datasets. The reason is that these locations in these two datasets are scattered more uniformly in the corresponding areas.

Finally, Fig. 17 depicts the average of AVC-SW's approximation errors for all three datasets and a sliding window of size 100. For each dataset, this is the average of the error $\bar{\varepsilon}$ measured for the Voronoi cell corresponding to each window

[9] http://geonames.usgs.gov/.

[10] http://www.rtreeportal.org/.

[11] We also performed the experiments by selecting this point out of the points in the dataset and the results were similar.

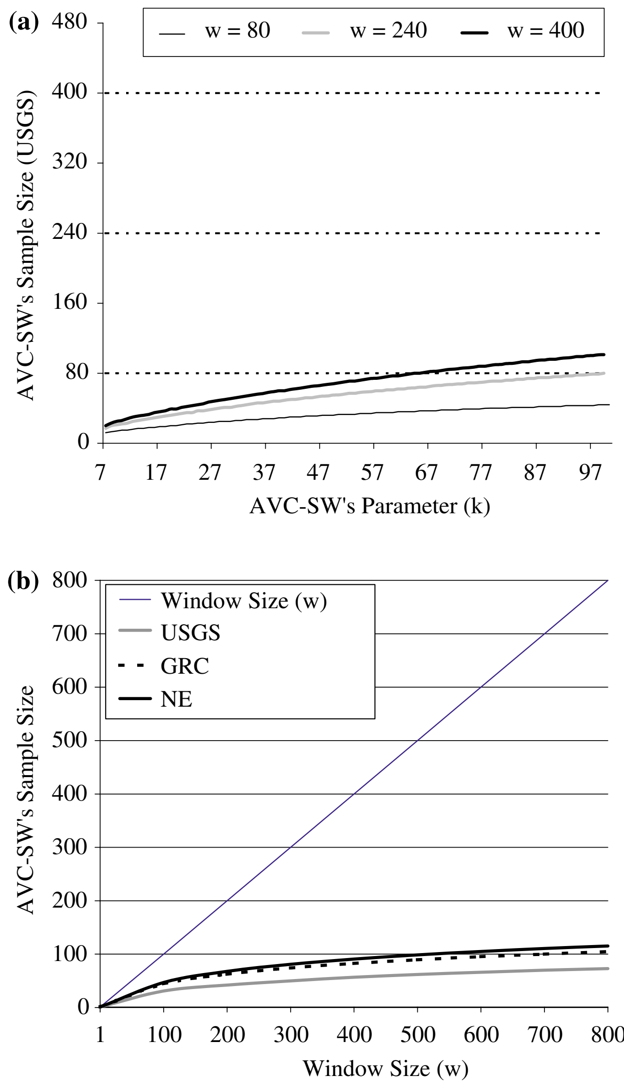

Fig. 16 Average number of points stored by AVC-SW (i.e., AVC-SW's sample size) for **a USGS** dataset and $w = 80, 240$, and 400 and different values of k, and **b USGS**, **GRC**, and **NE** datasets and different values of w when $k = 36$

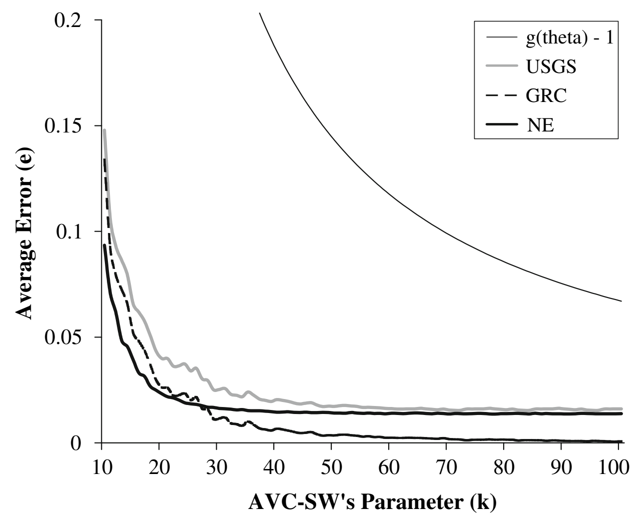

Fig. 17 Average approximation error of AVC-SW for **USGS**, **GRC**, and **NE** datasets when $w = 100$

over the dataset. The Figure also shows AVC-SW's analytically computed error $\varepsilon = g(\theta) - 1$ for different values of k (Theorem 3). As shown in the figure, the average error of all three datasets is considerably less than the analytical error ε. While this is true for the average case, we observed that the maximum error of all Voronoi cells computed is very close to ε. That is, AVC-SW's approximation error computed according to Theorem 3 is a tight upper bound for AVC-SW's actual error in real-world scenarios.

9 Related work

Query processing on spatial data streams has received recent attention by the database community [5,11,10] to solve problems in the spatial domains, which shows the potential challenges involved. Moreover, different variations of Voronoi diagrams have also been used as index structures for the nearest neighbor search [8,14,24]. However, to the best of our knowledge no other study has considered building Voronoi cells on spatial data streams. In this section, we study research studies in three different areas related to the focus of this paper.

9.1 General and spatial data stream processing

Generally, the field of computational geometry (CG) is a rich area of investigation for the data stream algorithms [18]. Recently, many studies have focused on CG problems on data streams. Indyk enumerates a current list in [13] and studies four fundamental problems in [12]. In [6], Feigenbaum et al. find the diameter and convex hull of a spatial data stream over a sliding window. Their sample uses $O(r)$ space with $O(r)$ and $O(\log r)$ processing time per point to maintain a $(1+O(1/r^2))$-approximation and $(1+O(1/r))$-approximation, respectively. In [5], Cormode and Muthukrishnan use the same sampling method as AVCs to build radial histograms and approximate a number of geometric aggregates such as diameter and furthest neighbor search on 2-d point data streams. In [11], Hershberger and Suri introduce adaptive sampling to maintain an approximate convex hull of spatial data streams with a distance of $O(D/r^2)$ from their exact convex hull, where D is the diameter of their sample set.

In parallel and independent from our work, Lin et al. [16] propose solutions for skyline queries on sliding windows over data streams. Upon the arrival of a new data point p, they drop all the old points dominated by p. While this policy reduces the memory requirements of their algorithm, they do not pro-

vide any theoretical bound on this reduction. Our AVC-SW algorithm utilizes a similar policy for which we provide theoretical bounds based on the parameter of the algorithm as well as the user's tolerable error.

9.2 Query processing based on Voronoi diagrams

In [8], Hagedoorn introduces a directed acyclic graph based on Voronoi diagrams. He uses the data structure to answer exact nearest-neighbor queries with respect to general distance functions in $O(\log^2 n)$ time using only $O(n)$ space. Stanoi et al. in [24] combine the properties of Voronoi cells (influence sets in their terminology) with the efficiency of R-trees to retrieve reverse nearest neighbors of a query point from the database. As a more practical example, Kolahdouzan and Shahabi [14] propose a Voronoi-based data structure to improve the performance of exact k-nearest neighbor search in spatial network databases. As another related study by the database community, Stanoi et al. in [23] partition the space around a given query point q into six equal sectors to retrieve q's reverse nearest neighbors from a database of points. While both the partitioning and the reverse nearest neighbor problem are related to our work, the contexts are different. We use *arbitrary* number of sectors to define our *sampling buckets* in a completely different *sliding window streaming* context. We utilize the general version of the partitioning in our sampling algorithms and provide *theoretical approximation errors* for k sectors.

9.3 Approximating Voronoi diagrams

Arya et al. [1] focus on approximating the Voronoi diagrams globally to answer ε-nearest neighbor queries. They build cells with the shape of hypercubes or the difference of two hypercubes. Har-Peled [9] partitions the space with an approximation of Voronoi diagrams. His space decomposition generates a compressed quadtree of size $O(n\frac{\log n}{\varepsilon^d}\log\frac{n}{\varepsilon})$ that answers ε-nearest neighbor queries in $O(\log(n/\varepsilon))$ time. Arya and Vigneron [2] have performed the only work on approximating Voronoi cells in d-dimensional space. Their approach combines the shape approximation and adaptive sampling techniques to build an approximate cell of size $O(1/\sqrt{\varepsilon})$ for d=2. They assume that the exact cell to be approximated is given. Then, they examine the Voronoi neighbors of the given point and the corresponding Voronoi vertices to keep the minimum number of Voronoi neighbors using which an ε-approximate cell for addressing nearest neighbor problem can be computed. This is the same as what we call a $(1+\varepsilon)$-approximate cell in this paper. This approach is not applicable to sliding windows over data streams as insertion/deletion of each single point might cause the sampling criteria to include or exclude a neighbor from the cell. This non-deterministic change results into storing all the points in the window.

10 Conclusion

We studied the problem of computing the Voronoi cell of a fixed point with respect to a sliding window over the spatial data stream of site points. To tackle the problem, we first developed AVC, a Voronoi cell approximation algorithm for the time series model. We theoretically computed the approximation error of AVC in terms of its single parameter. To focus on our main interest, we extended AVC which resulted into AVC-SW for approximating the Voronoi cell with the sliding window model. Our main findings are as follows:

- Both AVCs construct $(1+\varepsilon)$-approximations to the Voronoi cell. That is, for each point q inside the approximate Voronoi cell of a point p, q's distance to r, its closest site point on the stream, is less than its distance to p by at most a factor of $1+\varepsilon$ (i.e., $\frac{|qp|}{|qr|} \le 1+\varepsilon$).
- Using Theorem 2, the parameter k (or θ) of both AVCs can be computed from the user's tolerable error ε. This means that the memory requirements of our algorithms can also be minimized according to the user's error threshold. On the other hand, Eq. 21 can be used to compute the value of k using which reduces the required memory to $\kappa < w$ in the sliding model for a given window size w.
- We theoretically computed the expected sample size of AVC-SW in terms of its parameter k, the window size w, and a harmonic number dependent on both k and w (see Theorem 4). With the sliding window model and a random arrival of the site points, AVC-SW significantly reduces the space complexity of the classic algorithm from $O(w)$ to $O(k\log(\frac{w}{k}+1))$ regardless of the distribution of the points in the 2-d space.

Acknowledgment This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (PECASE), IIS-0324955 (ITR), and unrestricted cash gifts from Google and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Appendix: The Proof of Lemma 1

Proof The proof of this lemma sets a lower bound on the number of sectors used in AVC (i.e., $k > 6$). First, assume that $\theta < \pi/3$ and the point $q \in N$ is inside $V'(p)$. The proof is by contradiction. Let S be the sector containing q. It is clear that q cannot be the minimum point of S as $V'(p)$ is the Voronoi cell of p with respect to the minimum points in N. Hence, suppose m is the minimum point corresponding to

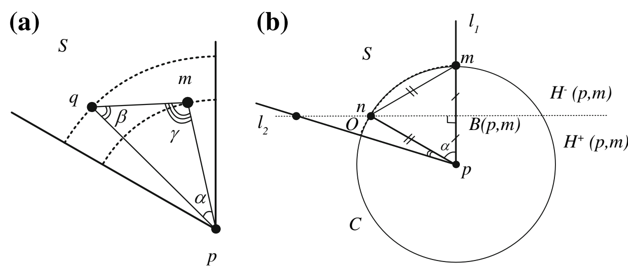

Fig. 18 The sector S and its minimum point m where **a** $\theta < \pi/3$, and **b** $\theta > \pi/3$

S (i.e., $m = m(S)$) as shown in Fig. 18a. Therefore, we have $|pq| > |pm|$. As q and m are both inside the same sector S, the angle between $\overline{pm}$ and $\overline{pq}$, $\alpha = \angle qpm$, is less than θ. Therefore, $\alpha < \pi/3$. In the triangle $\triangle pqm$, $|pq| > |pm|$ concludes that $\gamma > \beta$. Clearly, as α, β, and γ are the angles of the same triangle, thus $\alpha + \beta + \gamma = \pi$. At least one of the angles β or γ must be greater than $\pi/3$ as $\alpha < \pi/3$. The fact that $\gamma > \beta$ yields that $\alpha < \pi/3 < \gamma$. That is, in the triangle $\triangle pqm$, we have $|pq| > |mq|$ and whence q is closer to m than p. Therefore, the bisector of $\overline{pm}$, $B(p, m)$, would exclude q from $V'(p)$. This means that $V'(p)$ does not contain the point q which contradicts our assumption.

Now assume that the angle θ is greater than $\pi/3$. We show that there exist a set of potential points in N which are not excluded from $V'(p)$ by any of bisector lines corresponding to the minimum points. Figure 18b shows a sector S and its boundaries l_1 and l_2. Assume that the corresponding minimum point of S, m, is on l_1. The locus of all the points in the sector S which are removed from N because of the minimum point m is outside the circle C centered at p with a radius of $|pm|$. The bisector line $B(p, m)$ intersects with the circle C at point n. We show that this point is inside S. As n is on $B(p, m)$ we have $|mn| = |pn|$. Besides, as n is on the circle C, thus $|pn| = |pm|$. Therefore, the triangle $\triangle pmn$ is an equilateral triangle. Hence, the angle $\alpha = \angle mpn = \pi/3$. This yields that $\alpha < \theta$. Therefore, the point n is inside the sector S. Now consider the intersection of the sector S, outside of the circle C, and $H^+(p, m)$ (marked as O in Fig. 18b). As n is inside S, this intersection is not empty. The AVC algorithm removes any point in this intersection because of the minimum point m. However, the bisector line $B(m, p)$ does not exclude these points from the approximate Voronoi cell of p. For any point q in this area, if none of bisector lines corresponding to the minimum points of other sectors excludes q from $V'(p)$, q will be inside $V'(p)$.[12] □

[12] When $\theta = \pi/3$, n is the only point in O and on the edge of $V'(p)$ if no other bisector line excludes it from $V'(p)$.

References

1. Arya, S., Malamatos, T., Mount, D.M.: Space-efficient approximate voronoi diagrams. In: Proceedings of the 34th ACM Symp. on Theory of Computing (STOC), pp. 721–730 (2002)
2. Arya, S., Vigneron, A.: Approximating a voronoi cell. Tech. rep. (2003). HKUST-TCSC-2003-10
3. Banaei-Kashani, F., Shahabi, C.: SWAM: A family of access methods for similarity-search in peer-to-peer data networks. In: Proceedings of the Thirteenth Conference on Information and Knowledge Management (CIKM'04), pp. 304–313 (2004)
4. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Heidelberg (2000)
5. Cormode, G., Muthukrishnan, S.: Radial histograms for spatial streams. DIMACS TR 2003-11 (2003) (in press)
6. Feigenbaum, J., Kannan, S., Zhang, J.: Computing diameter in the streaming and sliding-window models. Algorithmica (2004) (in press)
7. Gardner, M.: The Sixth Book of Mathematical Games from Scientific American. University of Chicago Press (1984)
8. Hagedoorn, M.: Nearest neighbors can be found efficiently if the dimension is small relative to the input size. In: Proceedings of the 9th International Conference on Database Theory—ICDT 2003, Lecture Notes in Computer Science, vol. 2572, pp. 440–454. Springer, Heidelberg (2003)
9. Har-Peled, S.: A replacement for voronoi diagrams of near linear size. In: Proc. 42nd Annu. IEEE Sympos. Found. Comput. Sci., pp. 94–103 (2001)
10. Hershberger, J., Shrivastava, N., Suri, S.: Cluster hull: A technique for summarizing spatial data streams. In: Proceedings of 22nd IEEE Conf. Data Engineering, ICDE'06. IEEE Computer Society (2006)
11. Hershberger, J., Suri, S.: Adaptive sampling for geometric problems over data streams. In: Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. ACM (2004)
12. Indyk, P.: Algorithms for dynamic geometric problems over data streams. In: Proceedings of the Thirty-Sixth annual ACM Symposium on Theory of Computing, pp. 373–380. ACM Press (2004). doi:10.1145/1007352.1007413
13. Indyk, P.: Streaming algorithms for geometric problems. In: Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04) (2004). Invited Talk
14. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04) (2004)
15. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of data, pp. 201–212. ACM Press (2000). doi:10.1145/342009.335415
16. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: efficient skyline computation over sliding windows. In: Proceedings of the 21st International Conference on Data Engineering (ICDE'05), pp. 502–513. IEEE Computer Society (2005)
17. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
18. Muthukrishnan, S.: Data streams: algorithms and applications. Tech. rep., Computer Science Department, Rutgers University (2003)
19. Okabe, A., Boots, B., Sugihara, K., Chiu, S.N.: Spatial Tessellations, Concepts and Applications of Voronoi Diagrams, 2nd edn. Wiley, New York (2000)
20. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**(4/5), 464–497 (1996)

21. Sharifzadeh, M., Shahabi, C.: Supporting spatial aggregation in sensor network databases. In: Proceedings of the 12th ACM International Symposium on Advances in Geographic Information Systems, pp. 166–175 (2004)
22. Sharifzadeh, M., Shahabi, C.: Utilizing Voronoi cells of location data streams for accurate computation of aggregate functions in sensor networks. GeoInformatica **10**(1), 9–36 (2005)
23. Stanoi, I., Agrawal, D., Abbadi, A.E.: Reverse nearest neighbor queries for dynamic databases. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 44–53 (2000)
24. Stanoi, I., Riedewald, M., Agrawal, D., Abbadi, A.E.: Discovery of influence sets in frequently updated databases. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01), pp. 99–108. Morgan Kaufmann Publishers Inc. (2001)
25. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 443–454. ACM Press (2003). doi:10.1145/872757.872812