Monitoring Orientation of Moving Objects around Focal Points

Kostas Patroumpas and Timos Sellis

School of Electrical and Computer Engineering
National Technical University of Athens, Hellas
{kpatro,timos}@dbnet.ece.ntua.gr

Abstract. We consider a setting with numerous location-aware moving objects that communicate with a central server. Assuming a set of focal points of interest, we aim at continuously monitoring object orientations and hence detect situations where many objects get closer to or move away from any such site. Towards this goal, we propose a streaming approach that delegates part of the processing to objects, which relay positional updates upon significant deviations at their course. The central processor maintains the changing distribution of current object headings around each focal point and may issue alerts once it observes many objects moving along a direction (e.g., increased northbound traffic near the stadium). To efficiently answer such navigational queries, we introduce a novel access method that indexes object headings influencing a specific site. Furthermore, we extent this scheme to examine trajectory movements around sites over the recent past. Experimental results verify that this framework is able to cope with scalable numbers of objects at reduced communication cost, while offering instant notification of important trends along diverse directions for multiple focal points.

1 Introduction

Proliferation of location-based applications has led into efficient algorithms for processing typical continuous queries, such as range or k-nearest neighbor search [2, 5, 11], dealing with current coordinates of monitored objects (e.g., humans, vehicles, devices etc.). Still, less attention is given to observing evolving trajectories or mutable motion patterns, such as abrupt velocity variations or unexpectedly increasing concentration of objects in particular regions across time.

In this work, we turn our focus on studying movement from a navigational perspective, by examining significant changes in object headings. In navigation, the *heading* (a.k.a. *bearing*) of a moving object is its orientation, expressed as an angle from a known direction, usually north. By collecting heading information from streaming positional updates of numerous objects, it could be feasible to observe their mode of progression. But objects usually move at diverse directions amenable to sudden changes (e.g., turns), so perhaps no safe conclusion on movement patterns can be drawn from such a volatile variety of orientations. Even a fact like "40% of objects currently move eastbound" scarcely offers a valuable knowledge, as relevant objects may be located anywhere in the monitored area.

In our view, it seems much more useful to maintain the distribution of object orientations with respect to selected *focal points* or *sites* of interest, like terminal stations, sporting venues, traffic junctions etc. For each such site (say, a stadium), we would like to detect orientation trends *online* and also distinguish influencing objects that converge to or diverge from that site; e.g., whether a large number of vehicles are currently moving westbound and may be approaching the stadium soon. Typically for vehicles, ships, aircrafts etc., we implicitly assume that each object follows a consistent movement, hence is not arbitrarily displaced, but is moving towards –more or less– the same direction over a time interval.

In effect, we suggest a framework that acts like a constellation of radars (one per site), offering better insight along frequently-followed directions at progressively finer resolution. This mechanism can answer continuous *orientation-based queries* like "identify trucks bound for the port from the west at a distance less than 2km" or "issue an alert once a squadron of aircrafts are heading towards Athens from southeast over the past 10 minutes". To assist efficient evaluation of such requests, we propose a novel index structure that organizes the detection range of a specific site as a hierarchical tree. Influencing objects are assigned to tree nodes that represent sectors at gradually refined angles and extents. This index supports multiple orientation-based queries associated with a common focal point, each inspecting a diverse range and direction around it.

Since the entire mechanism must work in a streaming fashion to keep in pace with the bulk of incoming geospatial data, we adopt a collaborative scheme, where objects are capable of communicating with a central server and also have minimal processing capabilities to retain their recent positions and update their heading. A set of fixed focal points are allocated in the monitoring area; for each observation site, the server maintains the current distribution of headings based on the most recent status of objects detected within its area of interest. Reducing communication overhead is a major concern, so frequent positional updates referring to slight changes in objects' movement should better be avoided.

To the best of our knowledge, this is the first work on monitoring streaming orientations of moving objects. Our contribution can be summarized as follows:

– We introduce a novel spatiotemporal access method, namely PolarTree, which can effectively maintain object headings of interest to a given focal point.
– We propose a stream-based processing scheme that can provide real-time response to an important –yet largely neglected– class of navigational queries.
– We further extent this mechanism by employing sliding windows, practically examining the general heading for evolving portions of objects' trajectories.
– We evaluate empirically the robustness and efficiency of the framework with scalable numbers of moving objects and various settings for focal sites.

The remainder of this paper is organized as follows: Section 2 discusses fundamental concepts concerning focal points and object headings. Section 3 introduces the structure of PolarTree and presents its properties and operations. The processing framework for monitoring object headings is described in Section 4. Experimental results are reported in Section 5. Section 6 briefly reviews related work. Finally, Section 7 offers conclusions and future research directions.

2 Preliminaries

2.1 Scope of Focal Points

We assume a finite set $F = \{f_1, f_2, \ldots, f_n\}$ of stationary focal points (sites), which can monitor a large number of location-aware objects continuously moving on the 2-d Euclidean space $\mathbf{E}$. Each site $f_i \in F$ has a *focal scope* that represents its maximum range for detecting objects moving in its vicinity. In this setting, we assume that the scope of each focal point f_i practically translates into a circle $O(f_i, R_i)$ of a given radius R_i centered at the fixed location of f_i. In fact, every focal point specifies an *advanced range search*, aiming not just to observe objects inside its circular scope, but also to distinguish their orientations.

We do not assume any particular allocation of sites on plane $\mathbf{E}$, so they can be distributed randomly, evenly, but typically depending on the application. For instance, a traffic monitoring system may configure focal sites at major junctions along arterial roads and highways, while an environmental application may opt for observation points near wildlife habitats. Hence, the scopes of any two focal points $f_i, f_j \in F$ may intersect, signifying a common interest on area $O(f_i, R_i) \cap O(f_j, R_j) \neq \emptyset$. Each focal point may also designate a different radius, as depicted in Fig. 1. It may occur that $\mathbf{E}$ is not covered in its entirety, i.e., $\mathbf{E} \neq \bigcup_{i=1}^{n} O(f_i, R_i)$, meaning that some areas of $\mathbf{E}$ may not be monitored at all. Finally, we make no assumption on the total count n of sites, although we expect that a few hundred focal points of adequate scopes are more than sufficient to monitor a large geographical area (e.g., a city or a national park).

2.2 Object Headings and Focal Distances

Each moving object o is aware of its current timestamped location $\langle x, y, t \rangle$, where x, y are the coordinates (on plane $\mathbf{E}$) of a point position measured at time instant t. An object also knows its *heading* with reference to a previously recorded position $\langle x_0, y_0, t_0 \rangle$. That previous position of object o can be either its last recorded location or an *anchor point* representing its origin, a designated position or even a shifting location somewhere along its route (Section 4.3). Anyway, the heading signifies the direction of movement and can be represented as an angle θ with respect to a fixed direction; if this angle is measured from north it is commonly known as *azimuth*. For facilitating geometric calculations, in our model we measure headings counterclockwise with respect to the positive x-axis. This reflects the *slope* of the line segment that connects these two locations, expressed as an angle $\theta \in [0, 2\pi)$ on the trigonometric circle (as indicated for object i in Fig. 1). Formally:

$$\theta = \begin{cases} atan2(y - y_0, x - x_0), & \text{if } y \geq y_0 \\ atan2(y - y_0, x - x_0) + 2\pi, & \text{if } y < y_0 \end{cases}$$

In fact, we use the variant function $atan2$ instead of $arctan(\frac{y-y_0}{x-x_0})$, such that the calculated slope θ is also mapped to the correct quadrant of the trigonometric circle, thus signifying the direction of the vector from previous position to the current one. Since function $atan2$ takes values strictly in $(-\pi, \pi]$, we add the term

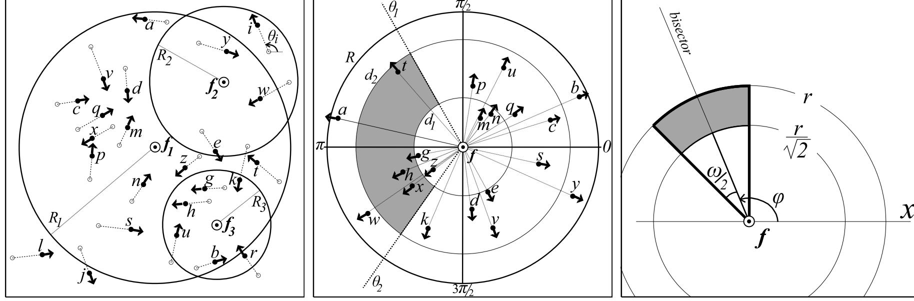

Fig. 1. Focal sites & scopes. **Fig. 2.** Polar mapping. **Fig. 3.** Polar sector.

2π for negative slopes, hence always $\theta \in [0, 2\pi)$. Figure 1 illustrates the current locations and headings of several objects moving in the vicinity of three focal points. Note that the heading of an object depends solely on its own movement from a previously recorded position and has nothing to do either with the focal point or the movement pattern of its neighboring objects.

Objects are moving freely, but we assume that their heading does not change abruptly at each positional update. Otherwise, had they been allowed to move towards random directions at each timestamp, there would be no reason in monitoring their incoherent orientations. Hence, objects are expected to follow a consistent course for a while, before heading towards another direction (e.g., making a turn). This motion pattern is frequently observed in several occasions of interest to location-aware applications, including vehicles, aircrafts, ships, migratory birds, etc.; so by no means is it limited to objects moving in fixed networks.

For an object o within the scope of a site f, its *focal distance* d is its Euclidean distance from f. Obviously, an object that influences multiple focal points, has different focal distances with respect to each one of them. For instance, object y in Fig. 1 is within the scope of both f_1 and f_2, but is currently closer to f_2.

2.3 Polar Mapping of Objects

To get better insight on the distribution of object headings around a given site f, we perform a mapping based on focal distance d and heading θ for every object o within the scope of f. As illustrated in Fig. 2 for the case of site $f \equiv f_1$, every qualifying object is mapped into a polar circle with center f (*pole*) and radius R (*scope*). Each object o is abstracted into a point at distance d from f and at an angle equal to its heading θ with respect to the positive x-axis. Locations beyond scope of f are ignored, as it happens for objects i, l, j, r in Fig. 1. Thus, each site f gets a clearer view of influencing objects and their distribution around f in terms of their focal distance and heading. We stress that this is not the usual mapping from Cartesian into *polar coordinates*, as our main concern is not just object positioning in relation to a specific point f, but their orientations around f instead. For example, the mapping of object t in Fig. 2 does not convey that t is at the northwest of nearby site f, but that t moves towards northwest.

Consequently, *orientation-based* queries related to focal sites can be expressed more easily. Formally, a query q related to a focal site f will search for objects:

$$\{o \mid o.heading \in [\theta_1, \theta_2) \wedge distance(o, f) \in (d_1, d_2]\}$$

i.e., all objects within the specified ranges for headings ($0 \leq \theta_1 \leq \theta_2 < 2\pi$) and focal distances ($0 < d_1 < d_2 \leq R$) from f. Intuitively, such a navigational query is transformed into a slice of a ring around site f (the shaded area in Fig. 2).

3 The PolarTree Index

In this section, we introduce a *main-memory* access method that can be used to index frequently updated object orientations observed around a single site f. A PolarTree partitions the focal scope of f into non-overlapping *polar sectors*, each denoting a specific range of object headings, since these are measured as angles. Then, movement directions (towards east, north, northwest, etc.) can be identified using suitable angular ranges. Recursive subdivision of the circular scope into smaller convex sectors, serves as the guiding principle for assigning objects at the nodes of this hierarchical tree, as exemplified in Fig. 4.

3.1 Index Structure

More specifically, the PolarTree is a binary tree with the following properties:

- The root node represents the entire scope R of focal site f and has no entries.
- Every node corresponds to a *polar sector* of a circle centered at f. Each polar sector is uniquely characterized by a *radius* r and its *bisector*, expressed as an angle ϕ on the trigonometric circle (Fig. 3). For instance, G is the only sector in Fig. 4a with radius $r = \frac{R}{2}$ and bisector at $\phi = \frac{\pi}{8}$.
- An internal node (i.e., not a leaf) with radius r has exactly two children, denoted as *leftChild* and *rightChild*, each with radius $\frac{r}{\sqrt{2}}$. The root (at level $l = 0$) has always two children, each with a radius equal to focal scope R.
- At an internal node, the central angle ω of its sector is bisected into two equal parts that characterize its children (Fig. 3). The size of angle ω is determined by the level of that node in the tree. For instance, nodes at level $l = 1$ are the children of the root and have $\omega = \pi$, nodes at level $l = 2$ have $\omega = \frac{\pi}{2}$ and so on. Therefore, the *angular range* $[\theta_{min}, \theta_{max})$ of each sector is unique, where $\theta_{min} = \phi - \frac{\omega}{2}$ and $\theta_{max} = \phi + \frac{\omega}{2}$. Consequently, the angular range[1] of its left child is $[\phi - \frac{\omega}{2}, \phi)$ and that of its right child is $[\phi, \phi + \frac{\omega}{2})$.
- Every node (excluding the root) maintains a catalogue of entries. Each entry denotes an object o with heading θ falling within the angular range of the respective sector and whose distance d from f is less than sector radius r. Object o is uniquely assigned to a single sector s, such that $\theta \in [s.\theta_{min}, s.\theta_{max})$ $\wedge\ d \leq s.radius$ and $\nexists\ s' \neq s,\ \theta \in [s'.\theta_{min}, s'.\theta_{max})\ \wedge\ d \leq s'.radius < s.radius$.

[1] The angular range is not actually an attribute of a node, as it can be easily calculated from its bisector ϕ and its level l in the tree (i.e., the size of angle ω). However, this notion is used in the sequel for better exposition of the algorithms.

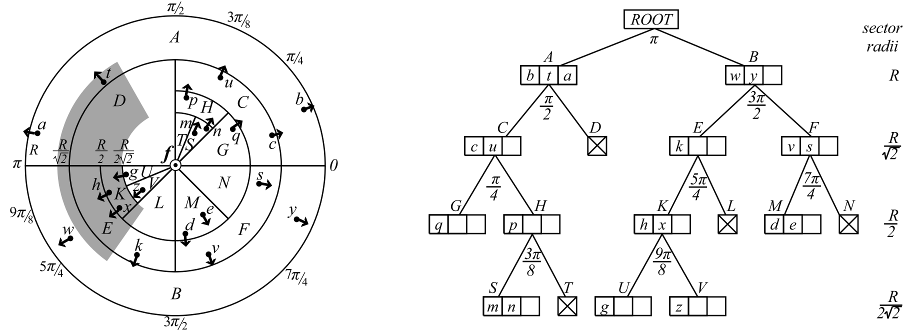

Fig. 4. (a) Headings assigned to polar sectors. (b) Polar tree with $M = 3$.

- Let M be the maximum number of entries allowed in a leaf node. A bisection occurs when a leaf has more than M entries (*overflow*), so it becomes an internal node itself with two new children. Entries of the original node may be assigned to its children, if their headings and distances from f qualify to the specifications of the new leaf nodes, as explained later on.
- Collapsing is applicable to leaf nodes only. If both descendants of an internal node are leaves and the total count of entries in these three nodes is less than a limit m (*underflow*), then the leaves are eliminated and their entries get merged with the entries of their parent, which now becomes a leaf itself.

Figure 4a illustrates the partition of the scope of focal point f for the example given in Fig. 2, assuming that at most $M = 3$ objects may be assigned at a leaf node. The shape of the corresponding PolarTree is shown in Fig. 4b, with the angles below each node signifying their distinctive bisectors ϕ.

Data is associated with both internal nodes and leaves (terminal nodes) of the PolarTree. Leaves stand for sectors originating from the focal point, while internal nodes practically represent *truncated sectors*. Indeed, an internal node with radius r represents a portion bounded by two "rings" at radii r and $\frac{r}{\sqrt{2}}$ around the focal point (shaded area in Fig. 3); in short, the area covered by its subtrees is cut off. Therefore, the focal scope is subdivided into non-overlapping portions, which are either truncated sectors assigned to internal nodes (e.g., sector F in Fig. 4) or circular sectors assigned to leaf nodes (e.g., sectors M, N).

By design, an internal node is responsible to monitor orientations occurring in the outer (truncated) part, while its descendants inspect the inner subsectors in more detail. Indeed, under a uniform distribution of headings, an object has equal probability to be monitored from a parent node or one of its descendants. So, a node and the unified set of its subdivisions purposely have equal shares of the focal scope, as illustrated in Fig. 3. Trivially, we can verify that:

Lemma 1. *For an internal node e in* PolarTree*, the circular area covered by both subtrees of e has size equal to the truncated sector assigned to node e.*

Leaf nodes may appear at any level $l \geq 1$, so the tree never collapses into a single leaf. In case that no object headings are currently found within the scope of f, its respective tree degenerates into a root node with two empty leaves corresponding to the two semicircles. A circular sector having no entries, remains as an empty leaf in the tree if its sibling is an internal node. For example, in Fig. 4, sector D exists as an empty leaf, since its sibling C is not a leaf.

Each entry is a tuple $\langle oid, addrH, fDistance, sign \rangle$, essentially pointing to the memory address $addrH$ where both the heading and location of object oid are actually maintained. Attribute $fDistance$ keeps track of the current focal distance of that object from site f, while $sign$ denotes whether the object converges to (+) or diverges (−) from f with respect to its previously known location. We stress that object headings and locations are maintained as items in a separate structure (e.g., an array $\mathbf{H}$ in our implementation), whereas entries of tree nodes just point to them. An object may possibly be found within the scope of multiple focal points, but its current location and heading are the same for each one of them. Hence, apart from simplifying the tree structure and reducing its space requirements, such a design decision makes a clear distinction between data concerning each individual object (all stored in array $\mathbf{H}$) and information referring to its influence on several focal points (maintained in separate PolarTrees).

3.2 Index Operations

The PolarTree index is inherently dynamic. Insertions and deletions may occur arbitrarily, while the tree always remains adjusted to assist searching for ranges of headings. Specifically, a PolarTree supports the following operations:

SearchSector. This search operation descends the PolarTree $\mathbf{T}$ of a focal point f in order to identify a sector s that corresponds to the given heading θ and focal distance d. Obviously, this sector s is unique among the contents of $\mathbf{T}$ and it is the strictest one that satisfies both $\theta \in [s.\theta_{min}, s.\theta_{max})$ and $d \leq s.radius$. Searching starts from the root and follows a single path, checking with the bisector and radius of each node to determine whether it should continue at one of its subtrees (Algorithm 1). Note that search may end up at an internal node, in case the specified heading and focal distance fall inside a truncated sector.

Insert. Insertions index object headings at suitable internal nodes or leaves of the PolarTree $\mathbf{T}$ built for a focal point f. First, *SearchSector* is invoked to identify sector s corresponding to the specified heading θ and focal distance d. Provided that such a sector exists (i.e. object o is not beyond the focal scope of f), a new entry for o is inserted into the catalogue of s. In case that s refers to a leaf and the new entry causes an overflow, procedure *Bisect* is invoked to split that leaf.

Delete. A deletion removes a given object o from the tree of site f. Again, *SearchSector* is called with the known heading θ and focal distance d of o to find its corresponding sector s. If s is a leaf, after removing o from its entries, a check for underflows is made by invoking operation *Merge* for the parent of s.

Bisect. Overflows are checked *for leaves only*, so a bisection does not affect upper tree levels. As already mentioned, it effectively partitions an existing sector s

Algorithm 1 PolarTree Operations

```
1: Function SearchSector (focal site f, angle θ, distance d)
2: Input: PolarTree T maintained for focal site f;
3: Output: the strictest sector s of T, s.t. θ ∈ [s.θ_min, s.θ_max) and d ≤ s.radius;
4: s ← T.root ;     //Initialize sector and start descending T following a single path
5: if s.radius < d then
6:    return nil;                //Beyond the scope of site f
7: end if
8: while s != nil do
9:    if s is leaf then
10:      return s ;
11:   else if θ < s.bisector and d ≤ s.radius/√2 then
12:      s ← s.leftChild ;       //Search left subtree
13:   else if θ ≥ s.bisector and d ≤ s.radius/√2 then
14:      s ← s.rightChild ;      //Search right subtree
15:   else
16:      return s;               //Search may end up at an internal node
17:   end if
18: end while
19: End Function
```

into three parts: a truncated sector that becomes an internal node and two new circular sectors as its children (leaves). Original entries of s are also partitioned, checking their focal distance and heading against the bounds of the three nodes.

Merge. This operation collapses two sibling leaves and appends their entries to their parent node, which becomes a leaf itself. As a precondition, the parent should not be the root, while the total count of entries at the three original nodes must be less than threshold m. In our setting, we have chosen $m = \frac{3}{4}M$ so as to avoid a possible bisection soon after a few subsequent insertions, but other values $m < M$ are also acceptable. Collapsing of leaves may propagate further up in the tree, as long as an underflow is discovered with respect to the new leaf node, its sibling (if also a leaf) and their parent node.

Update. To update the PolarTree $\mathbf{T}$ of site f with the current heading θ and focal distance d of an object o, we must first identify the sector s where o has been assigned before. As shown in Algorithm 2, an invocation to *SearchSector* is made with the previous heading θ' and distance d' of o (retrieved from array $\mathbf{H}$). In case that current values of θ and d fall beyond the bounds of sector s, then o must be removed from the catalogue of that node and should be inserted into a suitable node of $\mathbf{T}$ by invoking an *Insert* operation (Lines 10-12). Note that if this insertion fails, object o is surely beyond the scope of f. But if o remains in the same sector, then only its focal distance should be updated in the catalogue of s. During an update, it is also determined whether o gets closer or farther from f (attribute *sign*), by comparing its focal distances d' and d (Lines 5-9).

RangeSearch. This method offers response to orientation-based queries associated to site f that specify a range $[\theta_1, \theta_2)$ for headings and another $(d_1, d_2]$ on

Algorithm 2 PolarTree Operations (*continued*)

1: **Function** *Update* (focal site f, object o, angle θ, distance d, angle θ', distance d')
2: **Input:** PolarTree **T** maintained for focal site f;
3: **Output:** sector s of **T** where object o is assigned to;
4: $s \leftarrow SearchSector(f,\ \theta',\ d')$; *//Sector where o resides due to previous assignment*
5: **if** $d' \leq d$ **then**
6: o.sign $\leftarrow -$; *//o is moving away from f*
7: **else**
8: o.sign $\leftarrow +$; *//o is approaching f*
9: **end if**
10: **if** $\theta \notin [s.\theta_{min}, s.\theta_{max})$ **and** $d \notin (\frac{s.radius}{\sqrt{2}}, s.radius]$ **then**
11: Remove o from the catalogue maintained at s; *//o has moved into another sector*
12: $s \leftarrow Insert(f, o, \theta, d)$; *//Insert o into a suitable sector*
13: **else**
14: $o.fDistance \leftarrow d$; *//o remains in the same sector, but update its focal distance*
15: **end if**
16: **return** s;
17: **End Function**

18: **Procedure** *RangeSearch* (sector s, angle θ_1, angle θ_2, distance d_1, distance d_2)
19: **if** $s! =$ nil **then**
20: **if** $\theta_1 < s.bisector$ **and** $d_1 \leq s.radius/\sqrt{2}$ **then**
21: *RangeSearch* (s.leftChild, θ_1, θ_2, d_1, d_2); *//Search left subtree of s*
22: **end if**
23: **for** each object entry o in the catalogue of s **do**
24: **if** $\theta_1 \leq o.heading < \theta_2$ **and** $d_1 < o.fDistance \leq d_2$ **then**
25: Report o; *//o is a qualifying object at sector s*
26: **end if**
27: **end for**
28: **if** $\theta_1 \geq s.bisector$ **and** $d_1 \leq s.radius/\sqrt{2}$ **then**
29: *RangeSearch* (s.rightChild, θ_1, θ_2, d_1, d_2); *//Search right subtree of s*
30: **end if**
31: **end if**
32: **End Procedure**

focal distances, as mentioned in Section 2.3. Since many paths of the tree may be probably visited, this procedure (pseudo-code given in Algorithm 2) is called for the root node and recursively performs a depth-first search. When visiting an internal node that represents a sector s, the algorithm must decide whether to further descend to a subtree by comparing the bisector of s with the given angle range and also checking if d_1 is less than the radius of its children (Lines 20-22 and 28-30). When backtracking, any qualifying entries of a visited node s with headings and focal distances falling within the given ranges are reported as results (Lines 23-27). For the query specified with the shaded area in Fig. 4, nodes A, D, B, E, K, L will be visited (in that order). With a small variation, this method can also distinguish between objects approaching site f and those moving away from it, by simply checking their respective *sign* values.

3.3 Discussion

A PolarTree arranges all headings of interest to a focal point f into compact sectors, which can get recursively refined for better monitoring of movements closer to f. The initial subdivision of focal scope may not necessarily be carried out with the x-axis as *prime bisector* (at the root), but across any arbitrary direction. For instance, if headings were measured as azimuths or a focal point was mainly concerned with east- or west-bound orientations, then the y-axis should be used as prime bisector. We opted for a scheme with its prime bisector at angle $\phi = \pi$, because all derived subsectors are mapped to well-known portions of the trigonometric circle with obvious advantages on geometric calculations.

Bisection adheres to a repetitive pattern applied to all tree levels. This strategy decomposes the initial scope into finer partitions for progressively obtaining higher resolution of movements that occur closer to the focal point. The less the radius of a sector, the more segmented the scope across that direction, thus offering more detailed tracking of orientation trends. Besides, overflow threshold M represents the maximum capacity of leaves and reflects the level of detail prescribed for orientations close to the focal point. In effect, M specifies the *resolution* at which a focal point wishes to observe movements in its vicinity.

The tree is usually unbalanced, since a uniform distribution of headings and focal distances could be observed only rarely. For a skewed distribution, where most objects head towards certain directions, the respective sectors would be gradually subdivided at very tiny angles. Nodes may be unevenly filled, and even internal ones may occasionally be left empty. Even under a uniform distribution, larger-area sectors expectedly contain many more entries (far from site f) than a tiny sector monitoring a small range of headings in the close vicinity of f.

The height of the tree for a focal point f depends on threshold M, the number N of objects currently within scope, but also on their distribution around f. But:

Lemma 2. *A* PolarTree *has height at least* $\frac{1}{2}(1 + \log_2 \lceil \frac{N}{M} \rceil)$.

Proof. Apparently, the more uniform the distribution of headings around f, the lower the tree height. According to Lemma 1 and assuming a uniform distribution, the count of entries assigned to a subtree at level l is $1/4$ of those assigned to a subtree at $l - 1$, i.e., proportional to their respective area. So, it turns out that a leaf at level $l > 0$ has $\frac{N}{2^{2l-1}}$ entries. Due to uniformity, all leaves are at the same level, since branching factor is 2 and applies to all nodes. In order for a leaf not to be split, it suffices that $\frac{N}{2^{2l-1}} \leq M$, which yields the lower bound. □

4 Processing Streaming Orientations of Moving Objects

4.1 System Model

System infrastructure for processing orientations consists of a central server that communicates with numerous moving objects via a cellular network. A number of *base stations* are merely used for relaying messages between the server and objects located in their cell, so we shall ignore their role on data processing.

Each object o is identified by its *oid* and has enough resources to retain its current position $\langle x, y, t \rangle$ and to calculate its velocity, i.e., its speed v and heading θ. Normally, each object notifies the server about its *status* by sending a tuple $\langle oid, x, y, t, v, \theta \rangle$ at a specified frequency, i.e., every τ_0 time units. Duplicate, delayed or out-of-order status updates are not considered, so all messages stream synchronously into the server at a sequential pattern for each object.

But a status update should be sent instantly, once the heading or speed deviate significantly (e.g., a sudden slow down or a turn) from the values conveyed to the server with the latest message. We employ a simple detection method that utilizes two system-wide parameters λ and $d\theta$ specifying *thresholds* for acceptable deviations in speed and heading, respectively. In particular, if an object o changes its speed from v to v', an update should be sent if $|v - v'| > \lambda v$, denoting that the object accelerated or decelerated more than $\lambda\%$ compared to its previous speed. Similarly, the server must be notified if the heading of o changes from θ to θ' and it occurs that $|\theta - \theta'| > d\theta$. Such lightweight calculations can be performed by every object with negligible overhead, although a more sophisticated dead-reckoning [12] or threshold-guided strategy [7] could also be applied.

The server registers a set F of focal points and at each time instant t inspects movements in their scope according to streaming object statuses. Status information is retained in an array $\mathbf{H}$ for all objects, but not all statuses get updated concurrently, since some objects may report more frequently than others, depending on their motion pattern. At any rate, no object status can be more than τ_0 time units older compared to timestamp t of a newly received message.

A focal point f_i with scope at fixed radius R_i may be dynamically registered with the server and remains active for a duration of δ_i time units, until it gets eventually suspended and possibly resumed after some time. For instance, observation points at highways may be turned on at rush hours or switched off at night. A server-resident PolarTree is dedicated to retain the distribution of object orientations related strictly with a single f_i, so the server keeps track of $|F|$ separate trees. In effect, each pole f_i maintains at its own PolarTree a *"polar chart"* (Fig. 4a) that always reflects objects' movement as observed from the perspective of that particular f_i. At any given timestamp, the shape of this tree is independent of the order that headings were inserted or deleted.

Several continuous orientation-based queries may be specified at each focal point (Section 2.3). Without loss of generality, we assume that queries get activated when their corresponding pole f is registered. At any instant t, query evaluation (with *RangeSearch*) is based on current entries at the PolarTree of f.

4.2 Continuous Monitoring of Object Headings

In order to successfully maintain current object orientations around focal sites, the server operates at each execution cycle (i.e., distinct timestamp t) in two phases: (i) processing all status updates currently received from objects, and (ii) refreshing status for the remaining objects that did not send updates.

i) Update phase. Incoming statuses get processed one by one, in strict arrival order. Once the server receives such a message, it attempts to identify affected

Algorithm 3 Server Operations

```
1:  Function UpdateStatus (object o)
2:  Input: Server-resident array H of most recent status for all monitored objects;
3:  Output: the next time instant τ_e that o should send a status update;
4:  τ_e ← τ_0;                                    //Initialize refresh time to default value
5:  q ← H[o].location;   p ← o.location;      //Previous and current location of o
6:  c' ← gridHash(q);   c ← gridHash(p);   //Grid cells of last and current location
7:  candSites ← focal points with scopes overlapping cells c and c';
8:  for each focal point f ∈ candSites do
9:     R ← f.radius;                                              //Focal scope of site f
10:    h ← distance(f.location, q); d ← distance(f.location, p); //Focal distances for o
11:    if d ≤ R then
12:       if h ≤ R then
13:          Update(f, o, o.heading, d, H[o].heading, h);  //o already monitored by f
14:       else
15:          Insert(f, o, o.heading, d);  //Object o has just become of interest to site f
16:       end if
17:       τ ← Estimate time that o will fall beyond the scope of f;    τ_e ← min(τ, τ_e);
18:    else
19:       if h ≤ R then
20:          Delete(f, o, o.heading, d);    //Object o just gone outside the scope of f
21:       else if d < h then
22:          τ ← Estimate time that o could reach the scope of f;    τ_e ← min(τ, τ_e);
23:       end if
24:    end if
25: end for
26: Update H[o] with current heading and location of o;
27: if τ_e < τ_0 then
28:    return τ_e;  //Earliest time for renewal, as estimated from all sites affected by o
29: else
30:    return nil;  //No need to change default object settings
31: end if
32: End Function
```

focal points and update their PolarTree (Algorithm 3). But a single status update may influence multiple focal points, if that object is currently located within their intersecting scopes (e.g., object g in Fig. 1). To quickly identify affected sites, focal scopes are indexed with a regular *grid partitioning* of the entire area $\mathbf{E}$ into $c \times c$ square cells (Fig. 5). Each grid cell c_i maintains a list of pointers to every site with a circular scope intersecting cell c_i. As soon as an update arrives from object o, its location is hashed against the grid to identify the corresponding cell c_i, thus determining that only the subset $S \subset F$ of *candidate* focal points indexed at c_i need be probed. In Fig. 5, when object k sends update, sites $S = \{f_1, f_3\}$ should be examined, since their scopes overlap its (dark-shaded) cell.

However, a status update may also signify that object o has just fallen beyond the scope of a site f, so any reference to o in its respective PolarTree must be eliminated. Thus, any focal sites influenced by the previous status of o should be

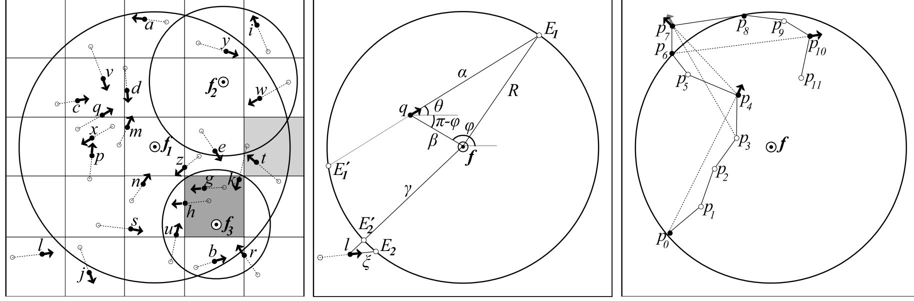

Fig. 5. Grid partitioning. **Fig. 6.** Forecasting. **Fig. 7.** Trajectory headings.

probed as well. By identifying the grid cell c' corresponding to the last known location of o, we get an additional set S' of candidate sites indexed at c' that also need examination. Figure 5 reveals that object k has just become of no interest to f_2, by only checking focal points $S' = \{f_1, f_2\}$ indexed at the light-shaded cell of the old location of k. Note that a site may appear in both S and S', in case that its scope overlaps with cells c and c' (which may be a single cell).

So it suffices that the status of object o is only checked against each candidate $f \in S \cup S'$ to detect changes in affected sites (Lines 8-25). There are four possible situations: (i) if o has just entered the scope of f, it must be inserted into its respective PolarTree, (ii) in case that o remains within scope of f, the PolarTree might need updating when o changes sector or its focal distance is modified, (iii) if o has just passed out of scope for f, then it must be removed from its tree, and (iv) if o stays beyond the scope of f, no further action is needed.

Besides, upon message receipt from an object at time t, the server also estimates the next time instant $t + \tau_e$ this object should relay its status again, so as to maintain a consistent distribution in relevant PolarTrees. We distinguish two cases that an object status should be renewed (Fig. 6), depending either on focal site(s) currently influenced or those that might soon be affected:

Departure Forecast. Let an object q be within the focal scope R of site f, and the server has just been informed for its current speed v and heading θ. Assuming that q continues the same course until further notice, the server is able to forecast when this object will fall out of scope R. Figure 6 depicts the expected course α of object q until it crosses the scope of f at location E_1. With simple trigonometric manipulations, it can be easily verified that $R^2 = \alpha^2 + \beta^2 + 2\alpha\beta\cos(\theta - \phi)$, where β is the focal distance of q and ϕ is the slope of segment β. This equation is always valid for all possible configurations of the heading and location of q at any quadrant inside the scope of f. It can be proven that the positive root α^+ corresponds to the distance from q to E_1, while the negative root α^- to E_1' (i.e., towards the opposite direction). Hence, after at most $\tau = \lceil \frac{\alpha^+}{v} \rceil$ time units, object q is expected to be found beyond the scope of f (Line 17).

Arrival Forecast. As shown in Fig. 6 for l, an object may not currently affect a site f, but it seems approaching; if l continues moving along ξ, it might soon

fall within scope at E_2. Yet, to safely predict the earliest time τ_e that l could cross the scope of any focal site, is not an easy task. Indeed, it could also involve inspection of sites $f \notin S \cup S'$, with scopes perhaps closer to object l, but indexed in cells neighboring to that of its current location. Sooner or later, l should send a status update (at most after τ_0 time units), so we opt for a simplified strategy that only examines candidate sites $f \in S \cup S'$ with negligible overhead. Given current focal distance γ and speed v of an object l, this approximation makes an eager forecast $\tau = \lfloor \frac{\gamma - R}{v} \rfloor$ of the time it takes to reach the scope of site f at E'_2, by ignoring actual heading as if l were directed straight towards f (Line 22).

Similar forecasts τ are made for all sites influenced by the currently examined object o. Among them, the smallest τ_e anticipates the earliest time that o may cause a change to current orientations (at time t) of any focal site. In case the interval τ_e is less than the prescribed renewal period τ_0, a message is sent to that object, specifying the time $t + \tau_e$ of its expected next update (Lines 27-31).

ii) Refresh phase. To maintain reliable object distributions around each site, the server *approximately* adjusts entries in PolarTrees for objects that have not currently relayed their status. Although velocity $\boldsymbol{v}$ of each such object o is deemed unchanged until further notice, its location $\boldsymbol{p}$ is not; hence, its focal distance from influenced sites changes. Assuming the known status of o is Δt units old, its expected position is $\boldsymbol{p} + \Delta t \cdot \boldsymbol{v}$; accordingly, the server rearranges entries in respective tree(s), in a fashion similar to the update phase. But, at this stage, no forecasting of status renewal times is made, so no messages are sent to objects.

4.3 Examining Trajectory Headings

Up to this point, processing only examines current object headings. However, it would be more insightful to monitor orientations referring to the most recent portion of every trajectory. With a *sliding window* of w time units that considers locations recorded during this evolving period, we can repetitively calculate for each object o its *trajectory heading* from the two extreme positions of o within that window, as indicated in Fig. 7 for $w = 5$ units and location updates every time unit. Such a setting does not actually change server-side processing, but it only modifies computation of headings separately at each object, provided they have enough memory to retain a finite portion of their own recent positions.

But for a realistic application, it seems more suitable to consider that sliding windows are specified along with focal points and their orientation-based queries. Hence, we assume that a sliding window w_i refers to a site f_i and is applied to all queries associated to f_i, thus affecting objects found within the scope of f_i with their headings computed from readings received during the past w_i time units (e.g., 10 min). Queries may specify various ranges for headings and focal distances, whereas an object may become of interest to multiple sites with diverse window extents. Thus, computation of trajectory headings has to be performed at the server, which should now retain a series of recent statuses for each object; still, objects send their status regularly, but also when a deviation is detected or in due time upon server request (displayed as black spots in Fig. 7).

A single status update from object o may cause changes to multiple trajectory headings maintained for o at diverse sites, because each f_i can specify a different window w_i, thus potentially returning a different anchor point. Each trajectory heading shall be derived from available object statuses received over the past w_i time units. Although not completely accurate, such an orientation still conveys the movement trend of every object, provided that $w_i > \tau_0$ to guarantee frequent renewal of all indexed headings. In Fig. 7, the trajectory heading at p_{10} will be correct due to availability of p_6, while the heading at p_7 will be slightly tilted by using p_4 as anchor point instead of non-relayed p_3. Obsolete statuses are purged from the server, when they cease to fall inside the window extent of any site.

5 Experimental Evaluation

Next, we report indicative results from an empirical validation of our framework for monitoring orientations. Due to space constraints, we refrain from discussing index performance and threshold calibration, and focus on server-side operations.

Experimental Setup. We generated synthetic datasets for varying numbers P of objects moving at diverse speeds along the road network of Athens (area ∼250 km^2). After calculating shortest paths between randomly chosen network nodes (i.e., origin and destination of objects), we took point samples at 200 concurrent timestamps along each such route. We also randomly selected diverse sets of n focal points at various radii R, which remain active all the time ($\delta = 200$ units).

All processing takes place in main memory. Algorithms were implemented in C++ and experiments with diverse parameter settings were simulated on an Intel Core 2 Duo 3GHz CPU running GNU/Linux with 2GB of main memory. Results are averages of actual measurements over 200 timestamps. Table 1 summarizes experimentation parameters and their ranges; default values are in bold.

System Configuration. For specifying granularity c of grid index for focal scopes, we measured the per cycle execution cost (sum of update and refresh times) for the most demanding case with 100k objects and diverse scope sizes for $n = 500$ sites (Fig. 8). As expected, grid partitioning proves more useful for larger scopes with higher degree of overlaps. We fix $c = 100$ in the sequel, as such a reasonably fine grid seems to offer better performance at all scope sizes.

Object-side parameters only control frequency of status updates. Next, we stipulate that objects should relay new status at most every $\tau_0 = 30$ timestamps, while we set thresholds $\lambda = 0.2$ and $d\theta = 30^o$, typically for moving vehicles.

Table 1. Experiment parameters.

Parameter	Values	Parameter	Values
Number P of objects	10k, 20k, 50k, **100k**	Grid granularity (c)	50, **100**, 500, 1000
Number n of sites	100, **200**, 500, 1000	Heading deviation ($d\theta$)	10°, 20°, $\mathbf{30^o}$
Focal radius (R)	0.5, 1, **2**, 3, 4 km	Speed deviation (λ)	0.05, 0.1, **0.2**
Leaf capacity (M)	100, **200**, 500	Window extent (w)	40, **50**, 100 units

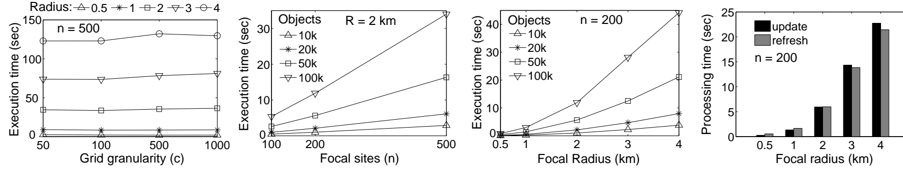

Fig. 8. **Fig. 9.** **Fig. 10.** **Fig. 11.**

Experimental Results. The main part of experiments refer to the efficiency and scalability of our approach. As shown in Fig. 9, the per cycle cost at the server depends on object count and is linear in the number of sites (their scopes fixed at 2km), since each PolarTree is maintained separately. But execution time escalates for larger scopes, as depicted in Fig. 10 for $n = 200$ sites, because the probability that an object influences multiple sites with intersecting scopes increases as well. This incurs additional overhead on forecasting and causes transmission of extra status updates from relevant objects, due to frequent crossing of scope boundaries. This is also verified from Fig. 11 that plots a breakdown of execution times per phase: handling incoming updates and forecasting arrivals and departures from focal scopes is often more costly than refreshing existing object statuses, especially for wider areas of interest. As it turns out, performance is sensitive to the size of scopes, but chiefly depends on their mutual overlaps. Nonetheless, for realistic radii (less than 3km) this scheme can always provide quick notification about observed orientations in less than 30 seconds.

Regarding communication cost, Fig. 12 illustrates the percentage of message savings for several scope sizes, i.e., the fraction of positional readings that did not cause any status change and hence were not relayed to the server. For small radii, the reduction in message transmission is considerable and may exceed 70%. But for larger scopes, the advantage of threshold-guided detection of motion changes is gradually annihilated, as an object becomes of interest to many sites and must report its status over and over due to their alternating demands.

In practice, focal scopes should be leveraged with appropriate choice of leaf capacity M. After all, it is improbable that a long-range site wishes to monitor movement trends at the finest resolution. As suggested in Lemma 2 and verified in Fig. 13, by increasing M the tree becomes shorter with wider sectors; a tree with 6 levels corresponds to central angles of 5.625^o at its bottommost leaves.

Concerning trajectory headings, they are handled exactly like current object orientations, but in addition require maintenance of sliding windows (Section 4.3). Figure 14 reveals that this maintenance overhead is proportional to scope size, but almost independent of the window extent. Anyway, such cost is negligible and can be compensated with valuable knowledge of recent orientations.

Finally, our processing scheme was designed to provide an approximate, yet consistent view of movements close to focal points. To assess the quality of such monitoring, we issued orientation queries based on polar sector boundaries (i.e., one query per polar sector). We then compared their approximate answers with

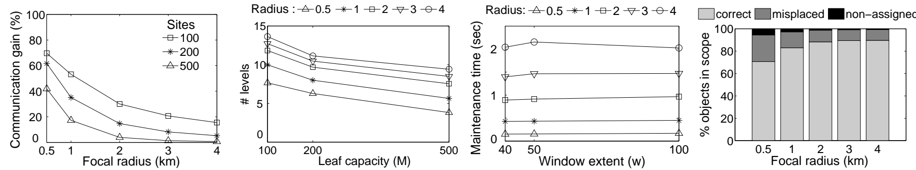

Fig. 12. **Fig. 13.** **Fig. 14.** **Fig. 15.**

those returned from an exhaustive evaluation where all objects relay their status at each timestamp, and thus always get mapped to correct sectors. In Fig. 15 the accuracy of answers is displayed for a single PolarTree (similar results obtained at multiple sites). At any time instant, less than 5% of qualifying objects are not reported within scope, but the majority of them (more than 70% at the worst case) are correctly assigned. Although monitored, another 15% of objects are misallocated to a neighboring sector due to small variations (that rarely exceed 8°) in their assumed heading. Indeed, smaller circles are subdivided in very tiny sectors, so it is more likely that an object be misplaced; yet, the wider the focal scope, the greater the accuracy of answers. Overall, polar charts prove able to offer a reliable insight into the actual distribution of object orientations.

6 Related Work

A taxonomy of spatiotemporal queries has been proposed in [8], distinguishing between *coordinate-based* queries, such as range or k-nearest neighbor search, and *trajectory-based* queries. This latter class includes *navigational* queries involving derived information of trajectories, like speed, heading, traveled distance etc. Index structures introduced in [8] aim at trajectory preservation, but no technique is suggested for maintaining object headings. Another type of spatial requests inspects *object-based directional* relationships [4, 10], e.g., identifying objects to the north of a given landmark. But such directional queries deal with relative positions of static features, and not with their movement and orientation.

In spatiotemporal databases, *dead-reckoning* policy suggests that an object should send a positional update when it deviates from its known motion vector, thus reducing communication cost. Two such schemes were introduced in [12] and adjust the uncertainty threshold at each update according to the current motion pattern. From a streaming perspective, in [7] we employed threshold-guided policies for online detection of movement changes in order to maintain concise trajectory synopses. All these approaches are orthogonal and can be easily integrated into our framework, as they only control object update frequency. Velocity vectors were also used in [3] to construct motion-sensitive bounding boxes for indexing moving objects. Although such structures can make predictions about future object positions, they are tailored for coordinate-based queries only.

Centralized or distributed techniques for managing streaming locations offer scalable techniques mostly for range [2, 6] or k-NN search [5, 11], by examining

only current object positions. We are not aware of other research work on processing object orientations in a streaming fashion. The proposed PolarTree is a hierarchical structure reminiscent of *space-driven* access methods for indexing multidimensional features [1]. Similarly to a quadtree [9], which is based on successive subdivision of areas into four equal-sized quadrants, a PolarTree utilizes *angle bisection* as its underlying design principle. Of course, our objective is not indexing locations of spatial features, but their changing orientations instead.

7 Concluding Remarks

In this paper, we have introduced a novel, simple, yet versatile, access method that can greatly assist continuous monitoring of movement orientations in suitably divided sectors around selected focal points of interest. We have also empirically evaluated the robustness and scalability of a processing scheme that offers real-time response to multiple requests with reduced communication cost.

In the future, we plan to study a variant tree structure with dynamic division in dissimilar sectors according to the observed density of object headings. Besides, distributed processing of orientations at designated base stations may further exhibit the powerfulness of the proposed spatiotemporal index.

References

1. V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2): 170-231, 1998.
2. B. Gedik, L. Liu. Mobieyes: A Distributed Location Monitoring Service using Moving Location Queries. *Transactions on Mobile Computing*, 5(10): 1384-1402, 2006.
3. B. Gedik, K.-L. Wu, P. Yu, and L. Liu. Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes. *IEEE TKDE*, 18(5): 651-668, 2006.
4. X. Liu, S. Shekhar, and S. Chawla. Object-Based Directional Query Processing in Spatial Databases. *IEEE TKDE*, 15(2): 295-304, 2003.
5. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *ACM SIGMOD*, pp. 634-645, June 2005.
6. S. Prabhakar, Y. Xia, D.Kalashnikov, W. Aref, and S. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10): 1124-1140, 2002.
7. M. Potamias, K. Patroumpas, and T. Sellis. Sampling Trajectory Streams with Spatiotemporal Criteria. In *SSDBM*, pp. 275-284, July 2006.
8. D. Pfoser, C. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *VLDB*, pp. 395-406, September 2000.
9. H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2): 187-260, 1984.
10. S. Skiadopoulos, N. Sarkas, T. Sellis, and M. Koubarakis. A Family of Directional Relation Models for Extended Objects. *IEEE TKDE*, 19(8): 1116-1130, 2007.
11. W. Wu, W. Guo, and K.-L. Tan. Distributed Processing of Moving k-Nearest-Neighbor Query on Moving Objects. In *ICDE*, pp. 1116-1125, April 2007.
12. O. Wolfson, P. Sistla, S. Chamberlain, Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, 7(3): 257-287, 1999.