Complex Motion Pattern Queries for Trajectories

Marcos R. Vieira
Supervised by Vassilis J. Tsotras
University of California, Riverside, CA USA
mvieira@cs.ucr.edu

***Abstract*— With the recent advancements and wide usage of location detection devices, large quantities of data are collected by GPS and cellular technologies in the form of trajectories. While most previous work on trajectory-based queries has concentrated on traditional range, nearest-neighbor and similarity queries, there is still the need to query trajectories using complex, yet more intuitive to users, motion patterns. In this paper, we describe several types of motion pattern queries for trajectories. In particular, we describe in detail two types of *motion pattern queries*: the *flexible pattern queries*, which focus on trajectories that follow a sequence of spatiotemporal events; and the *density-based pattern queries*, where the goal is to search trajectories that "stay together" for a long period of time. We then conclude this paper by briefly describing two other novel complex motion pattern queries that are currently under development.**

I. Introduction and Motivation

The wide availability of location and mobile technologies (cheap GPS devices, ubiquitous cellular networks, RFIDs, etc.), as well as the improved location accuracy (e.g. *A-GPS*, *E911*) has enabled many applications that generate and maintain data in the form of *trajectories*. A trajectory has a unique identifier and consists of location data (e.g. latitude/longitude) gathered for a specific moving object over an ordered sequence of time instants.

Past research efforts on querying trajectory data has mainly concentrated on traditional spatiotemporal queries, such as range and nearest neighbors searches (e.g. finding trajectories that passed by a predefined area), or similarity/clustering based tasks, such as extracting similar movement patterns and periodicities from trajectory data (e.g. finding all trajectories that are similar to a given query trajectory according to some similarity measure). Nevertheless, trajectories are complex objects whose behavior over space and time can be better captured as a sequence of interesting events. Only recently a few works have concentrated on "motion patterns" to query trajectories, which is the main focus of this research.

In the first part of this paper, Section II, we describe *flexible pattern queries* [12], [9], which allow users to select trajectories based on specific interesting events. Such patterns are described as regular expressions over a spatial alphabet that can be implicitly or explicitly "anchored" to the time domain. Moreover, it allows users to include "variables" in the query pattern, and thus greatly increase its expressive power.

We then describe in Section III *density-based pattern queries* [11], [8], which search for trajectories that follow a pattern that captures the "aggregate" behavior of trajectories as groups. Consider, for example, finding groups of trajectories that move "together", i.e. within a predefined distance to each other, for a certain continuous period of time. Such queries typically arise in surveillance applications, e.g., identify groups of suspicious people, convoys of vehicles, etc. We describe several strategies to discover such patterns in trajectorial data.

In Section IV, we conclude this paper by briefly describing two novel motion pattern queries that are currently under development as part of this research.

This research was partially supported by NSF IIS grants 0705916, 0803410 and 0910859, and a CAPES/Fulbright Ph.D fellowship.

II. Flexible Pattern Queries

Given the nature of trajectories as typically long sequences of events, a single range predicate may provide too many results (e.g., many trajectories passed through region A), while a similarity-based query may be too restrictive (e.g., not many trajectories match the full extent or large part of the query trajectory). Instead, here we propose a framework for processing *flexible pattern queries* over trajectories. Such queries combine the ability of fixed and variable predicates, with explicit or implicit temporal constraints and distance-based constraints.

A flexible pattern query specifies a combination of spatiotemporal predicates that can thus capture only the parts of trajectories that are of interest to the user. For example: "find all trajectories that first went by region A, then were closest to C, and ended up in E between 10pm and 11pm". This query simply provides a collection of range and Nearest-Neighbor (*NN*) conditions, as well as a explicit time constraint that all have to be satisfied in the specified order (implicit temporal constraint). Another predicate that can also be used to build very complex patterns is "variables" ("..., and they started and ended up by the same area in an interval of 10 hours apart."). Conceptually, flexible pattern queries cover the query choices between single predicates and similarity queries.

We note that patterns as effective ways to query data have been examined in the past. [2] examine patterns over event streams. Nevertheless, trajectories differ since they have both spatial and temporal behavior, which makes the work in [2] not efficient for querying trajectories. In spatiotemporal databases, patterns have been examined in [3], [5], but they concentrate on language/modeling related issues, providing less query support (e.g., no temporal and/or numerical constraints) and have less efficient/general evaluation methods.

$$\mathcal{Q} \rightarrow (\mathcal{S}\ [\bigcup \mathcal{D}])$$
$$\mathcal{S} \rightarrow \mathcal{S}.\mathcal{S}|P|!P|P^{\#}|P^{+}|P^{*}|?^{+}|?^{*}$$
$$P \rightarrow \langle op, \mathcal{R}[, t]\rangle,\ \mathcal{R} \in \{\Sigma \cup \Gamma\}$$
$$op \rightarrow disjoint|meet|overlap|equal|inside|contains|covers|coveredBy$$
$$\Sigma = \{A, B, C, ...\},\ \Gamma = \{@a, @b, @c, ...\}$$
$$t \rightarrow (t_{from} : t_{to})\ |\ t_s\ |\ t_r$$

Fig. 1. The Flexible Pattern Query Language.

A. Proposal of Solutions

We assume that the spatial domain is partitioned to a fixed set Σ of non-overlapping regions. Several levels of partitions can be created in order to define a hierarchy of regions (see Figure 2), where the user has the ability to define queries with finer alphabet granularity (*zoom in*) for the portions of greater interest and higher granularity (*zoom out*) elsewhere. Regions correspond to areas of interest (e.g. *school districts*, *airports*) and form the alphabet used in our query pattern specification. In the following we use capital letters to represent the region alphabet, $\Sigma = \{A, B, C, ...\}$.

A general pattern query $\mathcal{Q} = (\mathcal{S}\ [\bigcup \mathcal{D}])$, Figure 1, consists of a sequence of spatiotemporal predicates P, specified using regions from Σ, while $\mathcal{D}$ represents a collection of constraints and distance functions (e.g. *NN*). Modifiers can also be used with P, e.g., "P^{+}": one or more occurrences of P. Each spatiotemporal predicate $P \in \mathcal{S}$ is defined by $\mathcal{R}$, which corresponds to a predefined spatial region in Σ (fixed predicate) or a *variable* in Γ, the operator *op*, which describes the topological relationship that a trajectory and the spatial region $\mathcal{R}$ must satisfy over the optional time interval t.

A predefined region $\mathcal{R} \in \Sigma$ is explicitly specified by the user in the query predicate, e.g., A. In contrast, a *variable* denotes an arbitrary region in Σ and it is denoted by using symbols in $\Gamma = \{@a, @b, @c, ...\}$. A *variable* takes a single value (instance) from Σ (e.g. $@a$=C), but one can also specify the possible values of a *variable* as a subset of Σ (e.g., "any city district with museums"). Moreover, the same *variable* can appear in several different predicates of $\mathcal{S}$. This is useful for specifying complex queries that involve revisiting the same region many times. For example, a query like "$@x.?^{*}.B.@x$" finds trajectories that started from some region, then at some point passed by region B and immediately after they visited the same region they started from. Note that for our purposes, the wild-card "?" is also considered a variable; however it refers to *any region*, and not necessarily the same region if it occurs multiple times within a pattern.

Spatiotemporal predicates however cannot answer queries with constraints (e.g. NN type of queries). This is because topological predicates are binary and thus cannot capture distance based properties of trajectories. The $\mathcal{D}$ component is thus used to describe constraints among the *variables* used in the $\mathcal{S}$ part. One interesting kind of constraint is the distance-based constraint that can have the form ($AGGR(d_1, d_2, ...);\theta$). For example, consider the following query $\mathcal{Q} = \{A.?^{*}.B.@x.@y.C.?^{*}.@z,\ SUM(d_1, d_2) < 100,\ d_1 = d(@x, @y), d_2 = d(@z, E)$, which selects trajectories, among the ones that satisfy $\mathcal{S}$, that have the sum of the distance between regions $@x$ and $@y$ and the distance between $@z$ and a fixed region E less than 100 feet. Hence $\mathcal{D}$ contains a collection of distance terms $d_1, d_2, ...$, where term d_i represents the distance between two *variable* regions or between a *variable* region and a fixed one. In the above example, the aggregate *AGGR* and checking θ functions are, respectively, *SUM* and "< 100", but other functions can be used (e.g. *AVG*, *MIN* for the aggregate function, and *MIN*, *Top-k* for the checking function).

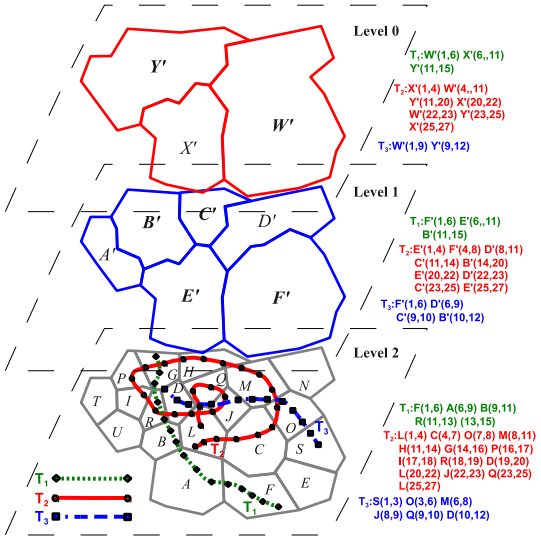

Fig. 2. Region-based trajectory representation.

We now proceed with the proposed index structures and algorithms used to efficiently evaluate *flexible pattern queries*. We use two lightweight index structures in the form of ordered lists, that are stored in addition to the trajectory data. There is one *region-list* per region and one *trajectory-list* per trajectory. The *region-list* $\mathcal{L}_{\mathcal{I}}$ of a given region $\mathcal{I}$, which does not have to be in Σ (see [12], [9]), acts as an inverted index that contains all trajectories that passed by region $\mathcal{I}$. Each entry in $\mathcal{L}_{\mathcal{I}}$ is a record that contains a trajectory identifier T_{id}, the time interval (*ts-entry*:*ts-exit*] during which the trajectory was inside $\mathcal{I}$, and a pointer to the *trajectory-list* of T_{id}. Records in a *region-list* are ordered first by T_{id} and then by *ts-entry*.

In order to fast prune trajectories that do not satisfy the query $\mathcal{S}$, each trajectory is approximated by the sequence of regions it visited. A record in the *trajectory-list* of trajectory T_{id} contains the region and the time interval (*ts-entry*:*ts-exit*] during which this region was visited by T_{id}, ordered by *ts-entry*. Note that entries in *trajectory-list* index point to their corresponding original trajectories in the trajectory archive. Given the index structures available, we propose four different strategies for evaluating flexible pattern queries:

1. *Index Join Pattern* (*IJP*): this method is based on a merge join operation performed over the *region-lists* corresponding to every fixed predicate in $\mathcal{S}$. The *IJP* uses the *region-lists* for pruning and the *trajectory-lists* for the *variable* binding (for more details, see [12]);

2. *Dynamic Programming Pattern* (*DPP*): this method performs a subsequence matching between the query pattern $\mathcal{S}$

TABLE I

EVALUATION OF FLEXIBLE PATTERN QUERIES.

P	Dataset	$\|\mathcal{S}\|$	$\|\mathcal{S}_f\|$	$\|\mathcal{A}\|$	*E-NFA*	*E-KMP*	*DPP*	*IJP*
$\mathcal{S}_1$	*Buses*	10	3	57	2.46	1.90	**1.11**	1.53
$\mathcal{S}_2$	*Buses*	20	7	29	89.62	62.75	28.99	**3.03**
$\mathcal{S}_3$	*Trucks*	20	7	76	111.91	54.68	30.28	**10.57**
$\mathcal{S}_4$	*Trucks*	46	29	11	3.06	0.73	**0.22**	1.56

(including *variables*) and the trajectory approximations stored as the *trajectory-lists*. The *DPP* uses mainly the *trajectory-lists* for the subsequence matching and performs an intersection-based pruning on the *region-lists* to find candidate trajectories;

3. *Extended-KMP* (*E-KMP*): this method is an extended version of the KMP method [3], which finds subsequence matches between the trajectory representations and the query pattern. The *E-KMP* contains extensions to handle the variable predicates (?, ?+), topological operations and implicit/explicit temporal constraints;

4. *Extended-NFA* (*E-NFA*): this method extends the work in [2] to cover topological operations, temporal constraints, and variables proposed in our language. This method, as well as the *E-KMP*, also performs an intersection-based pruning on the *region-lists* to fast prune trajectories that do not satisfy the fixed spatial predicates in S.

B. Results

We proceed with some experimental results that evaluate four flexible pattern queries using the *Buses* and *Truck* datasets [1]. For simplicity, we assume that the spatial domain is partitioned into regions using a uniform grid of 100×100 in a single level.

Since in these two real datasets trajectories move in relatively similar ways, we experimented with larger number of predicates so as to create more selective queries. Moreover, all queries contain between 2 and 4 variables and several wildcards $?^+$ and $?^*$. Table I shows the total number of predicates ($|\mathcal{S}|$), the number of fixed predicates ($|\mathcal{S}_f|$), the number of trajectories returned ($|\mathcal{A}|$) and running time (in seconds) for all four evaluation methods.

The results show that the *E-NFA* algorithm performs worse for all queries. This is because it cannot take advantage of the existing indexing structures so as to focus the search only on those parts of the trajectories that might contain answer (except from the original trajectory pruning using the *region-list* intersection). Our proposed two algorithms, *DPP* and *IJP*, have typically more robust behavior; nevertheless, *E-KMP* still shows competitive behavior for some queries. A thorough experimental evaluation with other datasets and query parameters can be found in [12], [9].

III. FLOCK PATTERN QUERIES

Recently, there has been increased interest in querying patterns capturing "collaborative" or "group" behavior in space and time between trajectories. This includes queries like *moving clusters* [7], *convoy queries* [6] and *longest flocks patterns* [4]. The difference between all those patterns is the way they define the relationship between the trajectories and their duration in time. Different from all the above definitions, here we consider the problem of identifying *all groups* of trajectories that stay "close" together for a given duration (not the longest as in [4]). Existing methods for flock pattern discovery [4] suffer from severe limitations. Such methods either find approximate solutions, or can be applied only for a single time instance of the problem (i.e. the solution does not support the minimum time duration in the query).

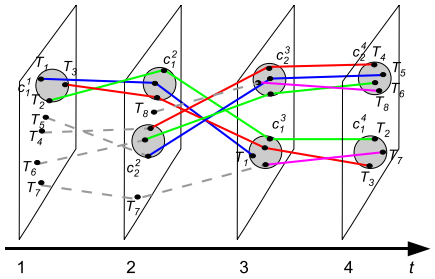

Fig. 3. Flock pattern examples: $\{T_1, T_2, T_3\}^{1-3}$, $\{T_4, T_5, T_6\}^{2-4}$.

We consider trajectories $\mathcal{T}$ to be part of a flock $\mathcal{F}$ if they are *all* within a maximum distance $\epsilon > 0$ to each other (i.e. if there exists a disk $c_k^{t_i}$ in time instance t_i and diameter ϵ covering *all* trajectories in $\mathcal{F}$ for a duration of δ consecutive time instants). A trajectory satisfies the above pattern as long as at least μ trajectories are contained inside the disk for the the time duration $\delta > 1$. The $c_k^{t_i}$ is called the center of the flock f_k at time t_i. Intuitively, a flock pattern can be viewed as a "tube" shape formed by the centers c and expanded with diameter ϵ in the space dimension, and having length δ in the time dimension, such that there are at least μ trajectories which stay inside the tube all the time. Figure 3 shows two examples of flock patterns for $\mathcal{F}(\mu$=3,ϵ,δ=3): $f_1 = \{T_1, T_2, T_3\}^{1-3}$ and $f_2 = \{T_4, T_5, T_6\}^{2-4}$.

A. Proposal of Solutions

The major challenge in evaluating flock pattern queries is to compute disks $c_k^{t_i}$. Since any point in the spatial domain can be a center of a flock, there is an infinite number of possible locations to test. Nevertheless, if we can find a disk $c_k^{t_i}$ with diameter ϵ that covers all trajectories in the flock f at time instance t_i, then there exists another disk with the same diameter but with different center $c'^{t_i}_k$ that also covers all trajectories covered by the first one and has at least two common points on its circumference.

In order to find the disk $c_k^{t_i}$, we use two trajectories' locations in t_i that have distance not greater than ϵ to each other. To efficiently find such pairs, we partition the spatial domain using a grid-based structure containing cells of size $\epsilon \times \epsilon$. We use several optimizations with this structure, which can be found in [11]. Using the above property to find disks $c_k^{t_i}$ for t_i, and the grid-based structure, we propose four strategies for evaluating flock pattern queries:

1. *Basic Flock Evaluation Algorithm* (*BFE*): this approach incrementally computes disks for t_i, and then joins them, using the $|c \cap f| \geq \mu$ joining condition, with disks previously found for t_{i-1}. The *BFE* reports a flock if there are at least δ join

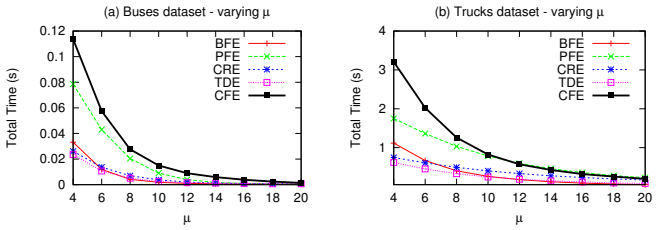

Fig. 4. Evaluating flock pattern queries with μ varying from 4 to 20.

consecutive operations applied over the same candidate set of trajectories, i.e. $u.t_{end} - u.t_{start}$=δ;

2. *Top Down Evaluation* (*TDE*): this method first selects candidate flocks by joining disks in t_i and t_δ. This is based on the assumption that the total number of candidate flocks generated using disks in t_i and t_δ is smaller than using disks in t_i and t_{i+1} (consecutive time instances). The set of candidate flocks still need to be further refined for t_i and t_δ. For this last refinement phase, the *BFE* method is used to evaluate each candidate flock in the candidate set;

3. *The Pipe Filter Evaluation* (*PFE*): this method employs the *filter-and-refine* paradigm. It first filters all trajectories that have at least μ objects within distance ϵ of them for a duration of at least δ time instances. Then, in a refinement step, for each candidate set, this method searches for flock patterns using the *BFE* method;

4. *The Continuous Refinement Evaluation* (*CRE*): this method uses the candidate disk generation step for time instance t_i as a filtering step to find candidate trajectories. These candidate trajectories are then analyzed in the next $t_{i-\delta}$ time instances, using the *BFE* method;

5. *The Cluster Filtering Evaluation* (*CFE*): this heuristic has two phases: (1) the *DBSCAN* clustering algorithm (*eps*=ϵ and *minPts*=μ) is used in each time instance t_i. This is similar to how *convoys patterns* are computed [6]; (2) then, each cluster found in t_i is further joined with clusters for t_{i-1}. If a cluster u can be augmented in this way for δ consecutive time instances ($u.t_{end} - u.t_{start} = \delta$), then the candidate trajectories in the cluster are analyzed using the *BFE* method.

B. Results

Figure 4 shows the average time (in seconds) to evaluate flock queries using ϵ=1.2, δ=10 and μ varying from 4 to 20. As it can be seen, when increasing μ, the average time needed to discover flock patterns for all methods decreases. This is expected since the flock queries become more selective and we have to maintain fewer candidate trajectories during the query evaluation. The number of flocks discovered range from 2,988 (μ=4) to 0 (μ=20) for the *Buses* dataset, and 14,935 (μ=4) to 309 (μ=20) for the *Trucks* dataset.

The *TDE* and *CRE* methods have significantly better performance compared to the other methods. The gap between those methods and the rest increases when the selectivity of the queries becomes low for small μ values. This is due to the large number of partial intermediate results which have to be maintained by the other two methods (*PFE* and *CFE*) and the increase of the total time needed to process those partial results. This is due to the fact that these two methods keep the trajectory history in a time window δ before computing the disks for each time instant.

The *CFE* algorithm has the worst performance among all methods. This is due to the fact that the filtering step in this approach employs clustering which can be very expensive for large datasets. This approach however works significantly better when the datasets are relatively small and the moving objects in those datasets have similar moving patterns. In scenarios like those, the cost for clustering is not that high which explains the improved performance.

IV. Conclusion and Current Works

In this paper we propose querying trajectories using *complex motion pattern queries*. In particular, we described in detail two interesting kind of motion pattern queries: the *flexible pattern queries* and *density-based pattern queries*.

As the next steps of this research, we are working on *pattern-based join queries*, which return pairs of trajectories that have *at least* a number of *fixed* or *variable* predicates in common. An example of such pattern query is "find pairs of trajectories that have at least 3 regions in common in the interval of 10 hours".

There are cases where a motion pattern query can easily lead to a vast number of trajectories in the result. Besides the result set having trajectories that all match the pattern query, they may also have other parts that are very similar to each other. Navigating through such a result set requires effort, and users give up after perusing through the first few answers. To address this problem, we are currently developing a framework to present the user with the most *diverse* trajectories among the answers. To define the diversity criteria among trajectories, we are currently exploring similarity-based functions. This framework is based on our previous work on query result diversification [10], where here we explore optimizations and heuristics specifically designed for trajectories.

References

[1] http://www.rtreeportal.org.
[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. *Proc. ACM SIGMOD*, 2008.
[3] C. du Mouza, P. Rigaux, and M. Scholl. Efficient evaluation of parameterized pattern queries. In *Proc. ACM CIKM*, 2005.
[4] J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *ACM GIS*, 2006.
[5] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. Tsotras. Complex spatio-temporal pattern queries. In *Proc. VLDB*, 2005.
[6] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1), 2008.
[7] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, 2005.
[8] M. R. Vieira et al. Characterizing dense urban areas from mobile phone-call data: Discovery and social dynamics. In *IEEE SocialCom*, 2010.
[9] M. R. Vieira et al. Querying spatio-temporal patterns in mobile phone-call databases. In *Proc. IEEE MDM*, 2010.
[10] M. R. Vieira et al. On query result diversification [accepted for publication]. In *Proc. IEEE ICDE*, 2011.
[11] M. R. Vieira, P. Bakalov, and V. J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *Proc. ACM SIGSPATIAL GIS*, 2009.
[12] M. R. Vieira, P. Bakalov, and V. J. Tsotras. Querying trajectories using flexible patterns. In *Proc. EDBT*, 2010.