On Discovery of Traveling Companions from Streaming Trajectories

Lu-An Tang[1,2], Yu Zheng[2], Jing Yuan[2,3], Jiawei Han[1]
Alice Leung[4], Chih-Chieh Hung[5], Wen-Chih Peng[6]
[1]University of Illinois at Urbana-Champaign; [2]Microsoft Research Asia
[3]University of Science and Technology of China; [4]BBN Technologies;
[5]Yahoo! Inc.; [6]National Chiao Tung University
{tang18, hanj}@illinois.edu, {yuzheng, v-jinyua}@microsoft.com
aleung@bbn.com, oshin@yahoo-inc.com, wcpeng@cs.nctu.edu.tw

Abstract— **The advance of object tracking technologies leads to huge volumes of spatio-temporal data collected in the form of trajectory data stream. In this study, we investigate the problem of discovering object groups that travel together (*i.e.*, *traveling companions*) from streaming trajectories. Such technique has broad applications in the areas of scientific study, transportation management and military surveillance. To discover traveling companions, the monitoring system should cluster the objects of each snapshot and intersect the clustering results to retrieve moving-together objects. Since both clustering and intersection steps involve high computational overhead, the key issue of companion discovery is to improve the efficiency of algorithms. We propose the models of closed companion candidates and smart intersection to accelerate data processing. A data structure termed *traveling buddy* is designed to facilitate scalable and flexible companion discovery from streaming trajectories. The traveling buddies are micro-groups of objects that are tightly bound together. By only storing the object relationships rather than their spatial coordinates, the buddies can be dynamically maintained along trajectory stream with low cost. Based on traveling buddies, the system can discover companions without accessing the object details. The proposed methods are evaluated with extensive experiments on both real and synthetic datasets. The buddy-based method is an order of magnitude faster than baselines. It also achieves higher precision and recall in companion discovery.**

I. Introduction

The technical advances in location-acquisition devices and tracking technologies have generated huge amount of trajectory data recording the movement of people, vehicle, animal and natural phenomena in a variety of applications, such as social network, transportation management, scientific studies and military surveillance: (1) In Foursquare [1], the users check in the sequence of visited restaurants and shopping malls as trajectories. In many GPS-trajectory-sharing websites like Geolife [35], people upload their travel or sports routes to share with friends. (2) Many taxis in major cities have been embedded with GPS sensors. Their locations are reported to the transportation system in the format of streaming trajectories [32], [28]. (3) Biologists solicit the moving trajectories of animals like migratory birds for their research [2]. (4) The battlefield sensor network watches the designated area and collects the movement of possible intruders [27]. Their trajectories are watched by military satellites all the time.

In the above-mentioned applications, people usually expect to discover the object groups that move together, *i.e.*, *traveling companions*. For example, commuters want to discover people with the same route to share car pools. Scientists would like to study the pathways of species migration. Information about traveling companions can also be used for resource allocation, security management, infectious disease control and so on.

Despite of the wide applications, the discovery of traveling companion is not efficiently supported in existing systems, partly due to the following challenges:

- Spatio-temporal co-location: Companions are objects that travel together. Here "travel together" means the objects are spatially close at the same time. Many state-of-the-art trajectory clustering methods, retrieving the object's major moving direction from their trajectories, ignore the temporal information of objects [20], [15], [23], [30], [33], [16]. Hence they cannot be directly used for companion discovery.
- Incremental discovery: In several applications like military surveillance, the system needs to monitor objects for a long time and discover companions as soon as possible. Hence the algorithm should report the companions in an incremental manner, *i.e.*, output the results simultaneously while receiving and processing the trajectory data stream.
- Efficiency: Most trajectories are generated in a format of data stream. Huge amounts of data arrive rapidly in a short period of time. The monitoring system has to cluster the data and intersect the clusters for companions. These steps involve high computational overhead. The algorithm should develop efficient data structures to process large scale data.
- Effectiveness: The number of companions is usually large. The system should report the large and long-lasting companions rather than small and short-time ones. The companion-discovery algorithm should be effective to select the most important results.

We are aware that several studies have retrieved object groups similar to the traveling companions, such as flock [12], convoy [17] and swarm [24]. However, most of them are designed to work on static datasets, some methods need multiple scans of the data, or cannot output results in an incremental manner. Hence it is still desirable to provide high-quality but less costly techniques for companion discovery on a trajectory stream.

In this study, we investigate the models, principles and methodologies to discover traveling companions from streaming trajectories. The main contributions of this paper include: (1) introducing the companion models to define the problem; (2) proposing the concepts of smart intersection and closed companion candidates to accelerate data processing; (3) analyzing the bottleneck of the problem and proposing a *traveling buddy* based approach; and (4) demonstrating the scalability and feasibility of the proposed methods by experiments on both real and synthetic datasets. Since the objects keep on moving in the trajectory stream, it is hard to maintain an index for their spatial positions. However, the relationship of objects are gradual evolutions rather than fierce mutations. The traveling buddies only store such relationship and can be easily maintained along the data stream. They can help discover companions without accessing the object details and significantly improve the algorithm's efficiency.

The rest of the paper is organized as follows. Section II defines the problem; Section III introduces the general framework of companion discovery; Section IV proposes the traveling buddy based method; Section V evaluates the algorithm's performances; Section VI carries out the discussions; Section VII gives a brief review of the related work and finally in Section VIII we conclude the paper.

II. Problem Definition

In the various applications of traveling companion, there are some common principles shared in different scenarios. We illustrate the characteristics of companion discovery by the following example.

Example 1: Ten objects are tracked by a monitoring system. Fig.1 shows their positions in four snapshots. There are three key issues to discover the companions:

- Cluster: The companions are the objects that travel together, *i.e.*, in the same cluster. Since the people, vehicles and animals often move and organize in arbitrary ways, the companion shape is not fixed. In Fig.1, the objects are grouped in round shape in snapshots s_1 and s_2, while in s_3, they are moving in a queue and the companions are formed as thin and long ellipses.
- Consistency: The companions should be consistent enough to last for a few snapshots. This feature makes it possible to find the companions by intersecting the clusters of different snapshots.
- Size: Most users are only interested in the object groups that are big enough. They may have requirements on the companion's size. For example, if the user sets the size threshold as four and requires the companion to last for at least four snapshots, then $\{o_1,\ o_2,\ o_3,\ o_4\}$ is the result companion.

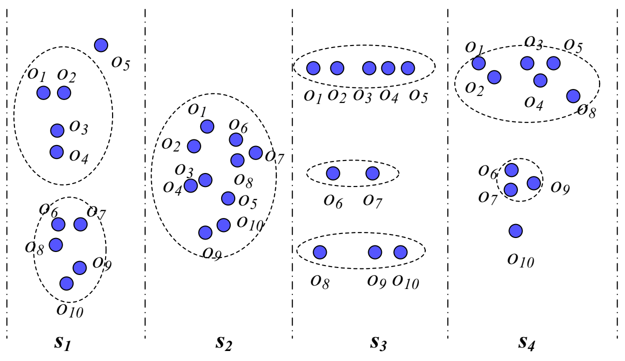


Fig. 1. Example: Discover Traveling Companions

To discover the traveling companions with various shapes, we employ the concepts of density-based clustering [8] in this study.

Definition 1 (Density Reachable): Let O be the object set in a snapshot, ε be the distance threshold, μ be the density threshold and $N_\varepsilon(o_i) = \{o_j \in O \mid dist(o_i, o_j) \leq \varepsilon\}$. Object o_j is directly density reachable from object o_i if $o_j \subset N_\varepsilon(o_i)$ and $|N_\varepsilon(o_i)| \geq \mu$.

Definition 2 (Density Connection): Let O be the object set in a snapshot, object o_i is density connected to object o_j, if there is a chain of objects $\{o_1,\ \ldots,\ o_n\} \in O$ where $o_1 = o_j$, $o_n = o_i$ such that o_{i+1} is directly density reachable from o_i.

With the concepts of density connection, we formally define the traveling companion as follows.

Definition 3 (Traveling Companion): Let δ_s be the size threshold and δ_t be the duration threshold, a group of objects q is called *traveling companion* if:
(1) The members of q are density connected by themselves for a period t where $t \geqslant \delta_t$;
(2) q's size $size(q) \geqslant \delta_s$.

Problem Definition: Let trajectory data stream S be denoted by a sequence of snapshots $\{s_1, s_2, \ldots, s_i, \ldots\}$. Each snapshot $s_i = \{(o_1, x_{1,i}, y_{1,i}), (o_2, x_{2,i}, y_{2,i}), \ldots, (o_n, x_{n,i}, y_{n,i})\}$, where $x_{j,i}$, $y_{j,i}$ are the spatial coordinates of object o_j at snapshot s_i. When the data of snapshot s_i arrives, the task is to discover companion set Q that contains all the traveling companions so far.

We will introduce the framework and techniques for companion discovery in the next few sections. Fig. 2 lists the notations used throughout this paper.

Notation	Explanation	Notation	Explanation
S	the trajectory stream	s, s_i, s_j	the snapshots in stream
C	the cluster set	c_i, c_j	the clusters
Q	the companion set	q	the traveling companion
R	the candidate set	r_i, r_j	the companion candidates
B	the buddy set	b_i, b_j	the traveling buddies
O	the object set	o_1, o_2, o_i	the objects
ε	the distance threshold	μ	the density threshold
δ_s	the size threshold	δ_t	the duration threshold
δ_γ	the buddy radius threshold	γ_i, γ_j	the buddy radius

Fig. 2. List of Notations

III. Companion Discovery Framework

A. The Clustering-and-Intersection Method

A general framework of *clustering-and-intersection* is proposed in [12], [17] to retrieve the convoy patterns. This framework can also be adapted to discover companions on trajectory stream: The idea is to retrieve *companion candidates* by counting common objects in the clusters from different snapshots. The system keeps clustering the objects in coming snapshots and intersecting them with the stored candidates. In this way the candidates are gradually refined to become resulting companions.

Definition 4 (Companion Candidate): Let δ_s be the size threshold and δ_t be the duration threshold, a group of objects r is a companion candidate if:
(1) The members of r are density connected by themselves for a period t where $t < \delta_t$;
(2) $size(r) \geqslant \delta_s$.

Intuitively, the companion candidates are the object groups with enough size but shorter duration. The candidate's size reduces when intersecting with the clusters from other snapshots, but its lasting time increases. Once a candidate's time grows longer than threshold, it will be reported as a traveling companion. Meanwhile, as soon as the candidate is not large enough, it is no longer qualified and should be removed from memory. Fig. 3 lists the steps of clustering-and-intersection algorithm.

Algorithm 1. Clustering-and-Intersection
Input: size threshold δ_s, duration threshold δ_t, distance threshold ε, density threshold μ, candidate set R and the trajectory data stream S
Output: every qualified companion q

```
1.  for each coming snapshot $s$ of $S$
2.      initialize new candidate set $R'$;
3.      cluster the objects in $s$ w.r.t to $\varepsilon$ and $\mu$;
4.      for each candidate $r_i \in R$, do
5.          for each cluster $c_j \in s$, do
6.              new candidate $r_i' \leftarrow r_i \cap c_j$;
7.              duration $(r_i')$ = duration $(r_i)$+duration $(s)$;
8.              if size$(r_i') \geq \delta_s$ then
9.                  add $r_i'$ to $R'$;
10.                 if duration $(r_i') \geq \delta_t$ then
11.                     output $r_i'$ as a qualified companion $q$;
12.     add all the new clusters to $R'$;
13.     $R \leftarrow R'$;
```

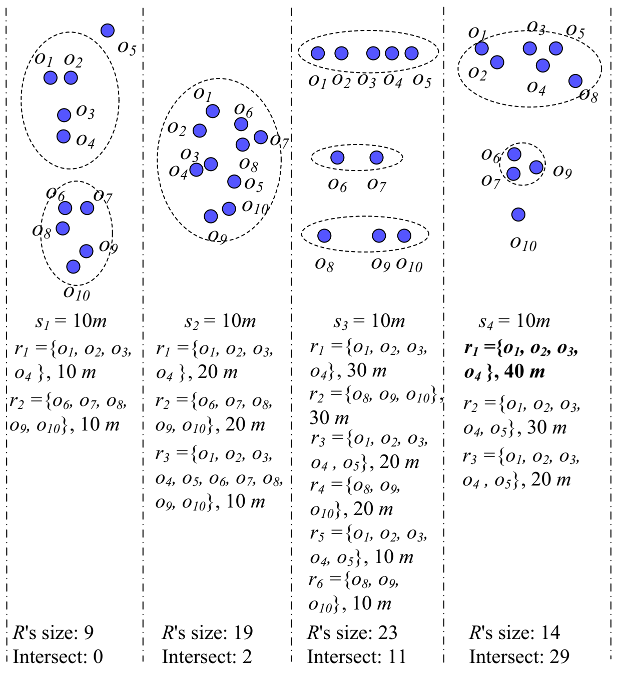
Fig. 3. Algorithm: The Clustering-and-intersection Method

Algorithm 1 first performs density-based clustering for all the objects in coming snapshot (Lines 1 – 3). Then the system refines companion candidates by intersecting them with new clusters (Lines 4 – 7). The intersection results with enough size are stored as new candidates (Lines 8 – 9). The ones with enough duration are reported as traveling companion (Lines 10 – 11). The new clusters are added to the candidate set (Line 12). At last the candidate set R is updated to process following snapshots (Line 13).

Proposition 1: Let n_1 be the size of objects and n_2 be the total size of candidate set R. The time complexity of Algorithm 1 is $O(n_1^2{+}n_1 * n_2)$.

Proof: In the clustering step, the algorithm needs $O(n_1^2)$ time to generate density-based clusters [1]. In the intersection step, suppose there are average m_1 clusters and m_2 candidates, the system carries out $m_1 * m_2$ intersections, and the intersection takes $l_1 * l_2$ time, where l_1 is the average cluster size and l_2 is the average candidate size. Since $m_1 * l_1 = n_1$, $m_2 * l_2 = n_2$, thus the time complexity of intersection step is $O(m_1 * m_2 * l_1 * l_2) = O(n_1 * n_2)$ and the total time complexity is $O(n_1^2{+}n_1 * n_2)$. ■

Example 2: Fig. 4 shows the running process of clustering-and-intersection algorithm. Suppose each snapshot lasts for 10 minutes, the size threshold is 3 and the time threshold is 40 minutes. The objects are first clustered in each snapshot. Two clusters in s_1 are taken as the candidates, namely r_1 and r_2. Then they are intersected with the clusters in s_2, meanwhile, the cluster of s_2 is also added as a new candidate r_3. The clustering and intersection steps are carried out in each snapshot. Finally, the algorithm reports $\{o_1, o_2, o_3, o_4\}$ as a traveling companion in s_4. The total intersection times are 29, and the largest candidate set R appears in s_3 with 23 objects involved.



o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10}
s_1 = 10m
r_1 ={o_1, o_2, o_3, o_4 }, 10 m
r_2 ={o_6, o_7, o_8, o_9, o_{10}}, 10 m
R's size: 9
Intersect: 0
s_2 = 10m
r_1 ={o_1, o_2, o_3, o_4 }, 20 m
r_2 ={o_6, o_7, o_8, o_9, o_{10}}, 20 m
r_3 ={o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}}, 10 m
R's size: 19
Intersect: 2
s_3 = 10m
r_1 ={o_1, o_2, o_3, o_4}, 30 m
r_2 ={o_8, o_9, o_{10}}, 30 m
r_3 ={o_1, o_2, o_3, o_4 , o_5}, 20 m
r_4 ={o_8, o_9, o_{10}}, 20 m
r_5 ={o_1, o_2, o_3, o_4, o_5}, 10 m
r_6 ={o_8, o_9, o_{10}}, 10 m
R's size: 23
Intersect: 11
s_4 = 10m
r_1 ={o_1, o_2, o_3, o_4 }, 40 m
r_2 ={o_1, o_2, o_3, o_4, o_5}, 30 m
r_3 ={o_1, o_2, o_3, o_4 , o_5}, 20 m
R's size: 14
Intersect: 29

Fig. 4. Example: The Clustering-and-intersection Method

[1]The clustering process can be improved to $O(n_1 * logn_1)$ with a spatial index, however it is costly to maintain such spatial index in each time snapshot [21].

B. The Smart-and-Closed Algorithm

The computational overhead of clustering-and-intersection method is high in both time and space. In each snapshot, the intersection is carried out in every pair of candidate and cluster. However, most intersections cannot generate qualified results with enough size. In this subsection we introduce the methods to improve the efficiency: (1) the smart algorithm stops the intersection step early once it is impossible to generate qualified candidates, and (2) the closed candidate are used to help reduce the memory cost.

Lemma 1: Let r be a companion candidate and δ_s be the size threshold, if there are more than $size(r) - \delta_s$ objects of r already appearing in intersected clusters, continuously intersecting r with remaining clusters will not generate any meaningful results with size larger than δ_s.

Proof: Since each object only appears once in a single snapshot and only belongs to one cluster[2], if there are more than $size(r) - \delta_s$ objects appearing in already intersected clusters, even in the best case (all the remaining objects are in a single cluster), the intersection result will still be smaller than $size(r) - (size(r) - \delta_s) = \delta_s$. ■

Lemma 1 can be used to improve the candidate refining process with smart intersection. Once an object is found in the cluster, the algorithm removes it from the candidate. The intersection process will stop earlier if there are less than δ_s objects remaining in the candidate.

Another problem of clustering-and-intersection method is the space efficiency, if all new clusters are added as candidates, the size of the candidate set will increase rapidly as trajectory stream passes-by, such a huge candidate set is a burden for system memory. In the worst case, all the clusters stay constant in the series of snapshots, the intersection process cannot prune any existing candidates and all the new clusters are added to the candidate set. After m snapshots, the system needs to maintain a $m * n$ size candidate set, where n is the number of objects.

In Fig. 4, candidates r_3 and r_5 in s_3 contain the same objects with different lasting time. In such cases, the system only needs to store the one with longer time (*e.g.*, r_3). Such candidates like r_3 are called *closed candidates*.

Definition 5 (Closed Candidate): For a companion candidate r_i, if there does not exist another candidate r_j such that $r_i \subseteq r_j$, and r_i's duration is less than r_j's duration, then r_i is a *closed candidate*.

Armed with Lemma 1 and Definition 5, we propose the smart-and-closed algorithm. The modifications are underlined in Fig. 5, the algorithm removes intersected objects from the candidate set and checks its remaining size before next intersection (Lines 5 and 9); when adding the new clusters to the candidate set, the algorithm always checks if there is already a candidate containing the same objects but with longer duration, only the ones passing the closeness check are added as new candidates (Lines 14 – 15).

[2]The clustering methods used in this study are all "hard-clustering", *i.e.*, an object can only belong to one cluster.

Algorithm 2. Smart-and-Closed Algorithm
Input: size threshold δ_s, duration threshold δ_t, distance threshold ε, density threshold μ, candidate set R and the trajectory data stream S
Output: every qualified companion q

```
1.  for each coming snapshot $s$ of $S$
2.      initialize new candidate set $R'$;
3.      cluster the objects in $s$ w.r.t to $\varepsilon$ and $\mu$;
4.      for each candidate $r_i \in R$, do
5.          for each cluster $c_j \in s$, do
6.              if $r_i$'s size is less than $\delta_s$ then break;
7.              new candidate $r_i' \leftarrow r_i \cap c_j$;
8.              duration $(r_i')$ = duration $(r_i)$+duration $(s)$;
9.              remove intersected objects from $r_i$;
10.             if size$(r_i') \geq \delta_s$ then
11.                 add $r_i'$ to $R'$;
12.                 if duration $(r_i') \geq \delta_t$ then
13.                     output $r_i'$ as a qualified companion $q$;
14.     for each cluster $c_j$ do
15.         if $c_j$ is closed then add to $R'$;
16.     $R \leftarrow R'$;
```

Fig. 5. Algorithm: The Smart-and-closed Discovery

In the worst case, Algorithm 2 cannot prune any candidates and the time complexity is the same as Algorithm 1. However, we find out that the smart-and-closed algorithm can save about 50% time and space in the experiments.

Example 3: Fig. 6 shows the running process of smart-and-closed algorithm. In snapshot s_3, when making intersections for candidate r_1 with three clusters, the process ends early after the first round. Since the system only stores closed candidates, the largest candidate set size is only 19 in s_2, and the total intersection time is 12, less than half of the cost in clustering-and-intersection.

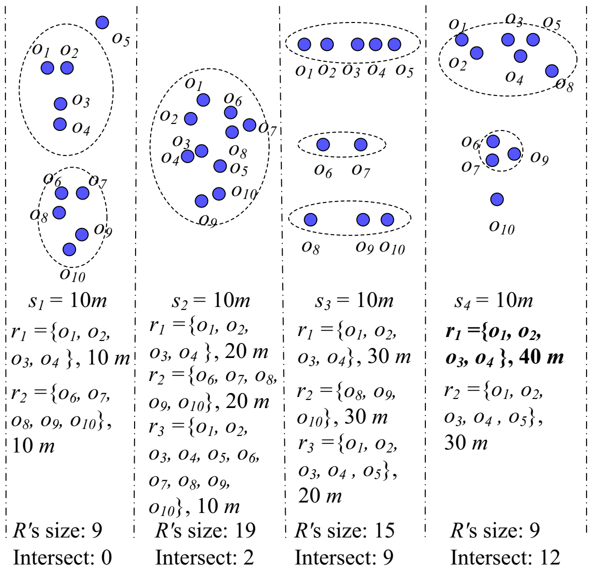

o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10}
s_1 = 10m
r_1 ={o_1, o_2, o_3, o_4 }, 10 m
r_2 ={o_6, o_7, o_8, o_9, o_{10}}, 10 m
R's size: 9
Intersect: 0
s_2 = 10m
r_1 ={o_1, o_2, o_3, o_4 }, 20 m
r_2 ={o_6, o_7, o_8, o_9, o_{10}}, 20 m
r_3 ={o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}}, 10 m
R's size: 19
Intersect: 2
s_3 = 10m
r_1 ={o_1, o_2, o_3, o_4}, 30 m
r_2 ={o_8, o_9, o_{10}}, 30 m
r_3 ={o_1, o_2, o_3, o_4 , o_5}, 20 m
R's size: 15
Intersect: 9
s_4 = 10m
r_1 ={o_1, o_2, o_3, o_4 }, 40 m
r_2 ={o_1, o_2, o_3, o_4 , o_5}, 30 m
R's size: 9
Intersect: 12

Fig. 6. Example: Smart-and-closed Algorithm

IV. Traveling Buddy based Discovery

Smart-and-closed algorithm improves the efficiency of intersection step to generate companions, but the system still has to cluster the objects in each snapshot. The density-based clustering costs $O(n^2)$ time without spatial index, where n is the number of the objects [14]. Due to the dynamic nature of streaming trajectories (*i.e.*, the objects' positions are always changing), maintaining traditional spatial indexes (such as R-tree or quad-tree) at each time snapshot incurs high cost [21]. In this section, we introduce a new structure, called *traveling buddy*, to maintain the relationship among objects and help discover companions.

A. The Traveling Buddy

In streaming trajectories, the objects keep on moving and updating their positions, however, the changes of object relationships are gradual evolutions rather than fierce mutations. The object relationships are possible to be retained in a few snapshots, *i.e.*, the objects are likely to stay together with several members of the current cluster. It is attractive to reuse such information to speed up the clustering tasks. However, the system cannot reuse it directly. The major issue is about the intrinsic feature of density-based clustering. Unlike other types of clusters, the results of density-based clustering may be quite different due to minor position change of an individual object. This phenomenon is called *individual sensitivity* as illustrated in Example 4.

Example 4: Fig. 7 shows two consecutive snapshots of the trajectory stream. Suppose the density threshold μ is set to three. In snapshot s_1, two clusters c_1 and c_2 are independent. However in s_2, object o_1 moves a little to the south, and this movement makes the two clusters density connected and merged as one cluster c_3. Such case may impose important meanings in real applications, for instance, in the scenario of infected disease monitoring, the people in the two clusters should then be watched together since the disease may spread among them.

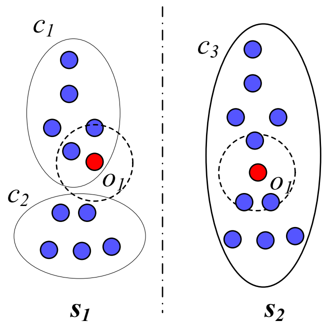


Fig. 7. Example: Individual Sensitivity Problem

The time cost of checking individual sensitivity is quadratic to the cluster size, and in many cases the system has to generate large clusters to produce meaningful companions. Hence high computational overhead is still involved in the clustering stage.

Then is it possible to explore a smaller and more flexible structure? In real world, there are some kinds of micro-groups in trajectory stream. For examples, couples would like to stay together on trips, military units operate in teams, families of birds, deer and other animals often move together in species migration. Such objects stay closer to each other than outside members. Even though they might not be as big as the companion, their information can be used to help clustering. Since they are way smaller than the cluster, their maintenance cost is much lower.

Definition 6 (Traveling Buddy): Let s be a snapshot of the trajectory stream and δ_γ be the buddy radius threshold, traveling buddy b is defined as a set of objects satisfying: (1) $b \subseteq s$; (2) for $\forall o_i \in b$, $dist(o_i, cen(b)) \leqslant \delta_\gamma$, where $cen(b)$ is the geometry center of b. The buddy's radius γ is defined as the distance from $cen(b)$ to b's farthest member.

The traveling buddies can be initialized by incrementally merging the objects in two steps: (1) treating all objects as individual buddies; and (2) merging them with their nearest neighbors. This process stops if the buddy's radius is larger than γ. The initialization step costs $O(n^2)$ time for n objects. However, this step only needs to be carried out once and the traveling buddies are dynamically maintained along the stream.

There are two kinds of operations to maintain buddies on the data stream: namely *split* and *merge*, as shown in the following example.

Example 5: Fig. 8 shows the traveling buddies in two snapshots. Traveling buddy b_1 is split into three parts in snapshot s_2. At the same time, b_2, b_3 and a part of b_1 are merged as a new buddy in s_2.

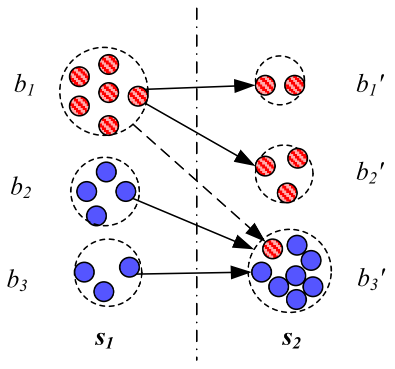


Fig. 8. Example: Merge and Split Buddies

When the data of a new snapshot s_{t+1} arrive, the maintenance algorithm first updates the center of each buddy b. For object $o_i \in b$, the system calculates the shift (Δx_i, Δy_i) between s_{t+1} and s_t. And the new center is updated as:

$$cen_{t+1}(b) = cen_t(b) + \sum_{o_i \in b} (\Delta x_i, \Delta y_i)$$

Then every object $o_i \in b$ checks its distance to the buddy center; if the distance is larger than δ_γ, o_i will be split out as a new buddy. The cen(b) is also updated by subtracting the shift of o_i.

The second operation is to merge the buddies that are close to each other. If two buddies b_i and b_j satisfy the following equation, they should be merged as a new buddy.

$$dist(cen(b_i), cen(b_j)) + \gamma_i + \gamma_j \leqslant 2\delta_\gamma$$

Suppose b_i has m_i objects and b_j has m_j objects, the new buddy b_k's center is computed as $cen(b_k) = (m_i * cen(b_i) + m_j * cen(b_j))/(m_i + m_j)$. Therefore, the system does not need to access the detailed coordinates of each object to merge buddies, the computation can be done with the information from the old buddy's center and size.

The detailed steps of buddy maintenance are shown in Fig. 9. When the data of a new snapshot arrives; the algorithm first updates the center of each buddy (Line 2). Then each buddy member is checked to see whether a split operation is needed (Lines 3 – 7). At last, the system scans the buddy set and merges the buddies that are close to each other. (Lines 10 – 13).

Algorithm 3. Traveling Buddy Maintenance
Input: the radius threshold δ_γ, the traveling buddy set B and the coming snapshot s
Output: updated buddy set B'

```
1.  for each $b_i$ in $B$ do
2.      update cen($b_i$);
3.      for $o_j$ in $b_i$, do
4.          if $dist(o_j, cen(b_i)) > \delta_\gamma$ then // Split Operation
5.              split $o_j$ out as a new buddy $b_j$;
6.              add $b_j$ to $B'$;
7.          update cen($b_i$);
8.      add $b_i$ to $B'$;
9.  //Merge Operation
10. for each $b_i, b_j$ in $B'$, $b_i \neq b_j$ do
11.     if $dist(cen(b_i), cen(b_j)) + \gamma_i + \gamma_j \leq 2\delta_\gamma$ then
12.         merge $b_i, b_j$ as $b_k$;
13.         remove $b_i, b_j$ and add $b_k$ to $B'$;
14. return $B'$;
```

Fig. 9. Algorithm: Buddy Maintenance

Proposition 2: Let m be the average number of traveling buddies and n be the number of objects. The time cost of Algorithm 3 is $O(n + m^2)$.

Proof: The split operation needs to check each object and the time cost is $O(n)$. The merge operation has to check the buddies pairs with time complexity $O(m^2)$. Therefore the total maintenance cost is $O(n + m^2)$. ■

In the worst case, if the objects are sparse and each of them is an individual buddy, where $m = n$. The maintenance cost is still $O(n^2)$. However the number of m is usually much smaller than n and the algorithm is likely to strike a relatively high efficiency.

B. Buddy-based Clustering

In the clustering step, the system has to check the density connectivity for each object. The traveling buddies can help the clustering process avoid accessing those object details. To bring down computational overhead, we introduce following lemmas.

Lemma 2: Let b be a traveling buddy, ε be the distance threshold and μ be the density threshold. If b's size is larger than $\mu+1$ and the buddy radius $\gamma \leqslant \varepsilon/2$, then all the objects in b are directly density reachable to each other. Such a traveling buddy is called a *density-connected buddy*.

Proof: Note that $\gamma \leqslant \varepsilon/2$, thus for $\forall o_i, o_j \in b$, $dist(o_i, o_j) \leqslant 2\gamma \leqslant \varepsilon$. Then all the members of b are included in $N_\varepsilon(o_i)$. If b's size is larger than $\mu + 1$, then $|N_\varepsilon(o_i)| \geqslant \mu$. By Definition 1, o_i and o_j are directly density reachable. ■

Lemma 2 shows that, if a traveling buddy is tight and large by itself, then all its members can be considered as density connected. Lemma 2 also gives the directions that the radius threshold δ_γ should not be set larger than $\varepsilon/2$.

Lemma 3: Let b_i and b_j be two traveling buddies with radius γ_i and γ_j, and ε be the distance threshold. If $dist(cen(b_i), cen(b_j)) - \gamma_i - \gamma_j > \varepsilon$, then the objects in b_i and b_j are not directly density reachable.

Proof: As shown in Fig. 10(a):
if $dist(cen(b_i), cen(b_j)) - \gamma_i - \gamma_j > \varepsilon$, then for $\forall o_i \in b_i, o_j \in b_j$, $dist(o_i, o_j) > \varepsilon$. Therefore, o_j does not belong to $N_\varepsilon(o_i)$ and they are not directly density reachable. ■

Lemma 3 tells us that, when searching for the directly density reachable objects for a traveling buddy, if another buddy is too far away, then the system can prune all its members without further computation. This lemma is very helpful. In the experiments it helps prune more than 80% of the objects.

For the traveling buddies that are close to each other, the detailed distance computation still needs to be carried out. But with the following lemmas, the system does not need to compute distances between all the pairs. Lemma 4 provides heuristics to speed up the computation.

Lemma 4: Let b_i, b_j be two density-connected buddies and ε be the distance threshold. If $\exists o_i \in b_i, o_j \in b_j$ such that $dist(o_i, o_j) \leqslant \varepsilon$, then all the objects of b_i and b_j are density connected.

Proof: As Fig. 10 (b) shows, since b_i is a density-connected traveling buddy and $|N_\varepsilon(o_i)| \geqslant \mu$, if $dist(o_i, o_j) \leqslant \varepsilon$, then o_i and o_j are directly density reachable. Since all the objects in b_i and b_j are directly density reachable from o_i and o_j, respectively. Therefore, all the objects in the two traveling buddies are density connected. ■

Based on Lemma 4, once the system finds a pair of objects close to each other, it ends the computation and considers the corresponding buddies density-connected. The detailed algorithm is listed in Fig. 11. The algorithm first updates the buddy set in a new snapshot (Line 1). Then it randomly picks a buddy and checks the density connectivity to others (Lines 2

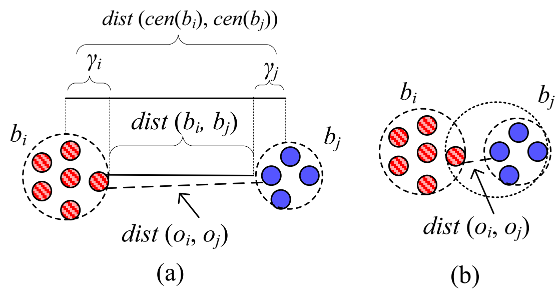


Fig. 10. Proof of Lemma 3 and 4

– 4). The far-away buddies are filtered out (Lines 5 – 6). With the help of Lemma 4, the algorithm searches density reachable buddies and objects and adds them to the cluster (Lines 7 – 13). Finally, the algorithm outputs clustering results when all the buddies are processed (Line 15).

Algorithm 4. Buddy-based Clustering
Input: the distance threshold ε, the density threshold μ, the coming snapshot s and the buddy set B.
Output: the cluster set C

```
1.  update buddy set $B$; //Algorithm 3
2.  randomly pick a buddy $b_i$ in $B$;
3.  initialize cluster $c_i \leftarrow b_i$, add $c_i$ to $C$;
4.  for $b_j$ in $B$, $b_j \neq b_i$, do
5.      if $dist(cen(b_i), cen(b_j)) - \gamma_i - \gamma_j > \varepsilon$, then
6.          continue; // Lemma 3
7.      for each $o_i$ in $b_i$, $o_j$ in $b_j$, do
8.          if $dist(o_i, o_j) \leq \varepsilon$, then
9.              if $b_i$ , $b_j$ are density connected then
10.                 add $b_j$ to $c_i$; //Lemma 4
11.                 break;
12.             else if $o_j$ is density reachable then
13.                 add $o_j$ to $c_i$;
14. repeat steps 2 - 13 until all buddies are processed;
15. return the cluster set $C$;
```

Fig. 11. Algorithm: Buddy-based Clustering

In the worst case, Algorithm 4 is still with $O(n^2)$ time complexity, where n is the number of objects. But in most cases, Lemmas 3 and 4 can prune majority buddies and save time for distance computation. The experiment results show that buddy-based clustering is an order of magnitude faster than the original clustering algorithm.

C. Companion Discovery with Buddies

The buddies are not only useful in clustering step, they are also helpful for the intersection process to generate companions. When intersecting a candidate with a cluster, the system needs to check whether each candidate's objects appear in the cluster or not. The information of traveling buddies can provide a shortcut to this process: If a buddy stays unchanged during the period, and it appears both in the candidate and the cluster, then the system can put all its members into the intersection result without accessing the detailed objects.

To efficiently utilize the buddy information, a buddy index is designed to keep the candidates dynamically updated with the buddies.

Definition 7 (Buddy Index): The buddy index is a triple $\{BID, ObjSet, CanIDs\}$, where BID is the buddy's ID, $ObjSet$ is the object members of the buddy, $CanIDs$ records the IDs of candidates containing the buddy.

As long as the buddy stays unchanged, the candidates only store the BID instead of detailed objects. While making intersections, the buddy is treated as a single object. When the buddy changes, the system updates all the candidates in $CanIDs$ and replaces BID with the corresponding objects in $ObjSet$. The buddy-based companion discovery algorithm is listed in Fig. 12.

Algorithm 5. Buddy-based Companion Discovery
Input: Size threshold δ_s, duration threshold δ_t, candidate set R, buddy index BI and the trajectory data stream S
Output: every qualified companion q

```
1.  for each coming snapshot $s$ of $S$;
2.      initialize new candidate set $R'$;
3.      buddy based clustering; // Algorithm 4
4.      update $BI$ and corresponding candidates;
5.      for each candidate $r_i$ in $R$, do
6.          if size($r_i$) < $\delta_s$ then break;
7.          for each cluster $c_j$ in $s$, do
8.              $r_i' \leftarrow$ buddy-based-intersection($r_i$, $c_j$);
9.              $duration(r_i') = duration(r_i) + duration(s)$;
10.             remove intersected objects and buddies from $r_i$;
11.             if size($r_i'$) $\geq \delta_s$ then
12.                 add $r_i'$ to $R'$;
13.                 if $duration(r_i') \geq \delta_t$ then
14.                     output $r_i'$ as a qualified companion $q$;
15.         for each cluster $c_j$ do
16.             if $c_j$ is closed then add to $R'$;
17.     $R \leftarrow R'$;
```

Fig. 12. Algorithm: Buddy-based Companion Discovery

When a new snapshot arrives, the algorithm performs buddy-based clustering and updates the buddy index (Lines 2 – 4), then selects out the candidates with enough size (Lines 5 – 6). The candidates are interested with the generated clusters with the help of the buddy index (Lines 7 – 10). The candidate's duration and size are checked again after the intersection, and the qualified ones are output as the companions (Lines 11 – 14). Finally, the closed candidates are added to the memory for further processing (Lines 15 – 17).

Example 6: Fig. 13 shows the running process for buddy-based companion discovery. There are four buddies initialized in snapshot s_1. In the candidates, the buddy ID is stored instead of detailed objects. In snapshot s_2, the four buddies stay the same and the algorithm makes intersections by only checking their BIDs. Although the total intersection time is not reduced, the time cost for each intersection operation has been brought down. It is common that different candidates

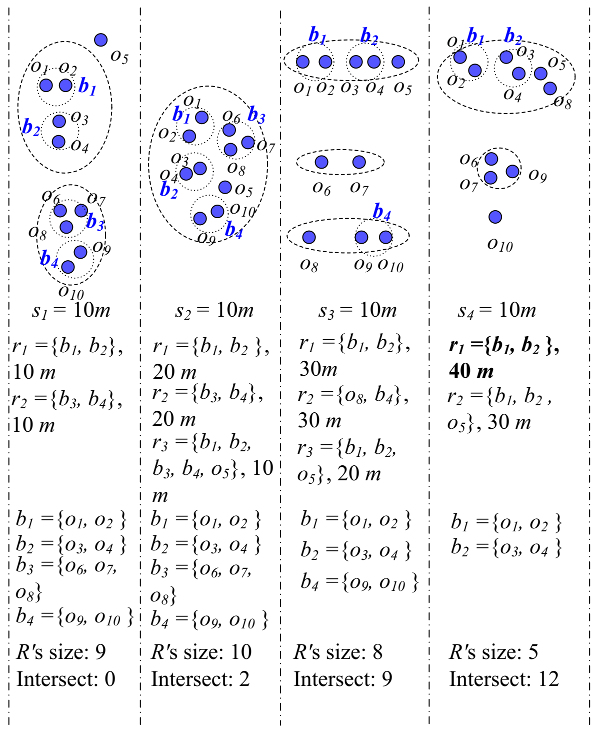


Fig. 13. Example: Buddy-based Discovery

contain the same objects, such as r_1 and r_3 in s_2. The buddy index helps to keep only one copy of the objects and add only pointers (the BIDs) to candidates. Therefore, the space cost is further reduced. In s_3, the buddy b_3 is no longer valid, then the system updates candidate r_2, using the objects to replace the buddy's ID. In s_4, traveling companion r_1 is discovered as $\{b_1, b_2\}$. With the help of buddy index, the system can easily look up detailed objects and output the companion as $\{o_1, o_2, o_3, o_4\}$.

V. Performance Evaluation

A. Experiment Setup

Datasets: We evaluate the proposed methods on both real and synthetic trajectory datasets. The taxi dataset (D_1) is retrieved from the Microsoft GeoLife and T-Drive projects [32], [35]. The trajectories are generated from GPS devices installed on 500 taxis in the city of Beijing. The dataset is available to public[3]. The military trajectory dataset (D_2) is retrieved from the CBMANET project [19], in which an infantry battalion of 780 units, divided as 30 groups, moves from Fort Dix to Lakehurst for a mission on two routes in 3 hours. Meanwhile, to test the algorithm's performance in large datasets, we also generate two synthetic datasets (D_3 and D_4), being comprised of 1,000 to 10,000 objects, with more than 10 million data records.

Baselines: The proposed Smart-and-Closed algorithm (SC) and Buddy-based discovery algorithm (BU) are compared with Clustering-and-Intersection method (CI), which is used as the framework to find convoy patterns [17]; and two state-of-the-art algorithms: (1) The Swarm pattern (SW) [24] that captures the objects moving within arbitrary shape of clusters for certain snapshots that are possibly non-consecutive; (2) The TraClu algorithm (TC) [20] that discovers the common sub-trajectories with a density-based line-segment clustering algorithm.

Environments: The experiments are conducted on a PC with Intel 6400 Dual CPU 2.13G Hz and 2.00 GB RAM. The operating system is Windows 7 Enterprise. All the algorithms are implemented in Java on Eclipse 3.3.1 platform with JDK 1.6.0. The parameter settings are listed in Fig. 14.

Dataset	Obj. #	Duration	Sample Freq.	Snapshot#	Record#
Taxi (D_1)	500	4.2 hours	5 minutes	50	25,000
Military (D_2)	780	3 hours	1 minute	180	140,400
Syn 1 (D_3)	1,000	24 hours	1 minute	1,440	1.44 M
Syn 2 (D_4)	10,000	24 hours	1 minute	1,440	14.4 M
The companion size threshold δ_s: 5 – 40, default 10					
The companion duration threshold δ_t: 3 – 15, default 10					
The clustering parameter ε and μ are set according to different datasets.					
The buddy radius threshold δ_γ: $\varepsilon/2$– $\varepsilon/10$, default $\varepsilon/2$.					

Fig. 14. Experiment Settings

B. Comparisons in Discovery Efficiency

In this subsection we conduct experiments to evaluate the efficency of companion discovery algorithms. Since both SW and TC cannot output the results incrementally, we take the running time of entire dataset as the measure for time cost. The size of candidate set (# of objects) is used to measure the space cost of companion computation. The only exception is TC, since the algorithm only carries out the sub-trajectory clustering task and does not store any companion candidates, TC's space cost is not included in the experiment.

We first evaluate the algorithm's time and space costs on different datasets with default settings. Fig. 15 shows the experiment results. Note that the y-axes are in logarithmic scale. BU achieves the best performances on all the datasets. In the largest dataset D_4, BU is an order of magnitude faster than CI and SW. BU's space cost is only 20% of SW and less than 5% of CI.

Figure 16 illustrates the influences of companion size threshold δ_s in the experiments. The experiment is carried on dataset D_3. Based on default settings, we evaluate the algorithms with different values of δ_s. Generally speaking,

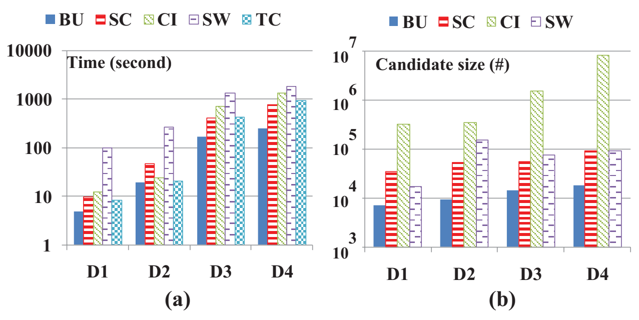


Fig. 15. Efficiency: (a) time, (b) space on diff. datasets

[3]GeoLife GPS Trajectories Datasets. Released at: http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/default.aspx

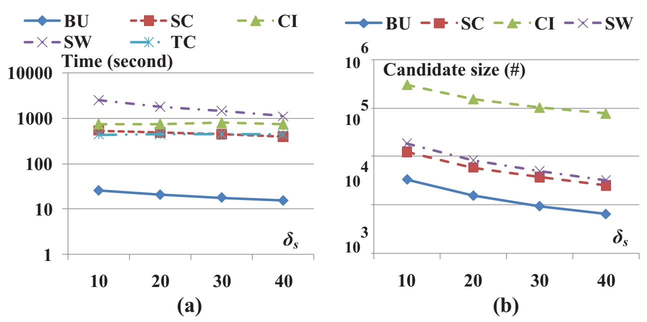


Fig. 16. Efficiency: (a) time, (b) space vs. δ_s

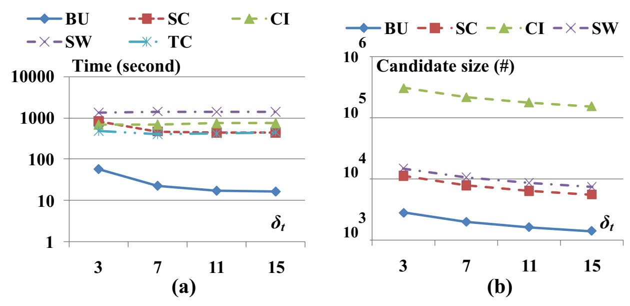


Fig. 17. Efficiency: (a) time, (b) space vs. δ_t

when the size threshold grows larger, the filtering mechanism is more effective to prune more companion candidates in each snapshot. The space costs reduce significantly, and the running times also decreases for fewer intersections.

We also study the influence of duration threshold δ_t. Based on default settings, the experiments are conducted on dataset D_3. The value of δ_t is changed from 3 to 15, the algorithm's performances are shown in Fig. 17. BU, SC and CI are all faster when δ_t grows larger, because many companion candidates are not consistent enough to last for a long time. When setting δ_t as 15 snapshots, BU can process the dataset in less than 20 seconds (Fig. 17 (a)). It is almost an order of magnitude faster than SC and CI. TC is not influenced by δ_s and δ_t, since it is only a clustering algorithm and does not generate any companion candidates. Beside TC, SW also could not improve the performance when δ_t increases, the reason is SW utilizes the *object-growth* strategy to prune candidates. Such heuristics could only work with the size threshold δ_s, but cannot benefit from larger δ_t.

C. Efficiency Analysis for Buddy-based Discovery

Why is the buddy-based discovery algorithm more efficient? BU outperforms other methods in the efficiency evaluations, especially in the scenarios of long lasting stream with large number of objects. In this subsection we carry out the experimental analysis to reveal the advantages of buddy-based discovery method.

In the beginning, we tune the parameters of BU to study the factors that influence its efficiency. With δ_s and δ_t set as default values, we test BU with different buddy radius threshold δ_γ from $\varepsilon/10$ to $\varepsilon/2$, and record the average buddy size $|b|$, buddy number and algorithm's running time. Their relationships are demonstrated in Fig. 18. One can clearly

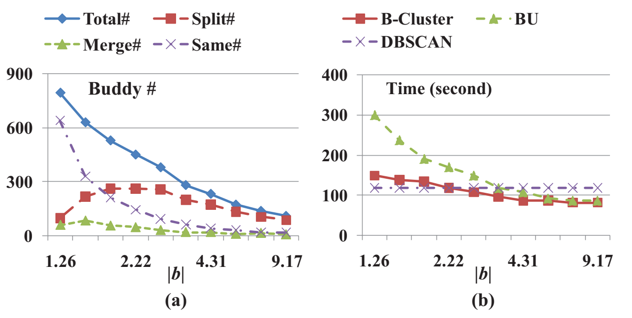


Fig. 18. Efficiency Analysis: (a) buddy number, (b) time vs. buddy size

learn from Fig. 18 (a) that the total buddy number is inversely proportional to the average buddy size $|b|$. In addition, the number of unchanged buddies decreases rapidly as $|b|$ grows larger. However, as shown in Fig. 18 (b), the running time of both buddy-based clustering (B-Cluster) and BU decreases with larger $|b|$. This phenomenon can be explained by Proposition 2, the cost of buddy's maintenance algorithm is $O(n + m^2)$, where n is the number of objects and m is the number of buddies. If n is fixed, then m is inversely proportional to $|b|$. Hence BU costs less time if $|b|$ is larger. Based on the efficiency analysis, we recommend setting the buddy radius as a relatively large value (such as $\varepsilon/2$). Fig. 18 (b) also records the time cost of DBSCAN clustering algorithm as a reference. Even if less than 20% buddies stay unchanged (which is rare for real-world objects), as long as the average size of the buddies is larger than 3, the buddy-based clustering algorithm can still outperform DBSCAN. The experiment results show that BU is especially feasible for processing a trajectory stream with dense object clusters.

BU has three steps, namely the maintenance step (M-step, Algorithm 3), clustering step (C-step, Algorithm 4) and intersection step (I-step, Algorithm 5). To study the time cost of each step, the system carries out BU on the four datasets and record the time costs of each step, as well as their proportions in the total running time, as shown in Fig. 19. The results denote that the clustering step is actually the most efficient in the three, it costs less than 5% of the total running time, compared to the *DBSCAN* clustering which usually takes 40-50% of the total running time of SC. BU spends an extra 10% to 15% time in maintaining the buddies to save more time from the clustering task. The two lemmas, especially Lemma 3, utilize the buddy information to filter out many objects without accessing their details. In addition, the buddy index helps to reduce the size of the candidate set, and so decreases the intersection times of companion discovery.

D. Evaluations on Algorithm's Effectiveness

The third part of the experiment is to evaluate the quality of the retrieved companions. In dataset D_2, an infantry battalion of 780 units moves from Fort Dix to Lakehurst for a mission on two routes in 3 hours. The objects are organized in 30 teams, each team has 25 to 30 units. The information of team partitioning is retrieved as the ground truth. The algorithm's outputs are matched to the ground truth and the measures of precision and recall are calculated as follows.

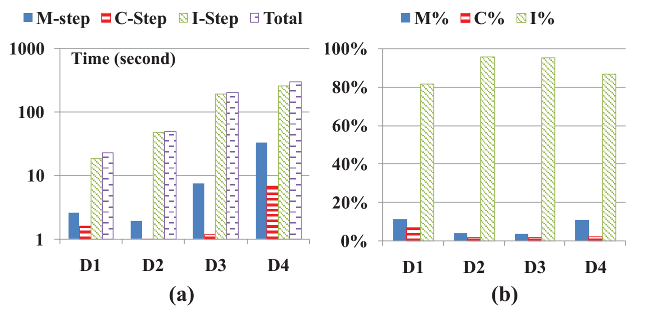


Fig. 19. Efficiency Analysis: (a) running time, (b) percentage of BU steps on diff. datasets

Precision: The proportion of true companions over all the retrieved results of the algorithm. It represents the algorithm's selectivity in finding out meaningful companions.

Recall: The proportion of detected true companions over the ground truth. This criteria shows the algorithm's sensitivity for detecting traveling companions.

We conduct experiments with different values of the size threshold δ_s. The results of effectiveness evaluation are shown in Fig. 20. BU and SC have same precision and recall since they output identical companions. They have about 20% precision improvement over SW, and near 40% precision improvement over CI. SW generates the swarm patterns of frequently meeting objects, which is actually a super set of the companions. The swarm pattern is highly sensitive to help find out all the companions (*i.e.*, 100% recall), but SW also generates more false positives that bring down the algorithm's selectivity. CI has the same problem with even lower precision. Since there are many redundant and non-closed companions in the results, more than half of CI's results are not useful.

Again, TC is not affected by the parameters of δ_s and δ_t. TC takes the movement direction as an important measure to compute sub-trajectory clusters; its results reflect the major directions of the object movements. However, such clusters may not capture the information of companions, because the companion member's moving direction might be different. As an illustration, please go back to Fig. 1. From snapshot s_2 to s_3, the moving directions of o_8 and o_9 are different, hence they may be put in different sub-trajectory clusters.

Another interesting observation is that, in Fig. 20, BU, SC, CI and SW's precisions all increase when δ_s becomes larger, since fewer companions can pass a higher size threshold. However, if δ_s is set too high (more than 25), several true companions will also be filtered out and the algorithm cannot achieve 100% recall.

Finally we study the influence of time threshold δ_t. Fig. 21 shows the precision and recall of the five algorithms with different δ_t on D_2. BU and SC achieve better performance than SW and CI. When increasing δ_t, the algorithm's precision increases, but they can still keep a high recall. Since all the true companions last for a long period in D_2. If we set δ_t greater than 11, both BU and SC can achieve 100% precision and recall. Hence we suggest that in real applications, the user should set a relatively high time threshold to filter out false positives, but a moderate size threshold to guarantee the algorithm's sensitivity.

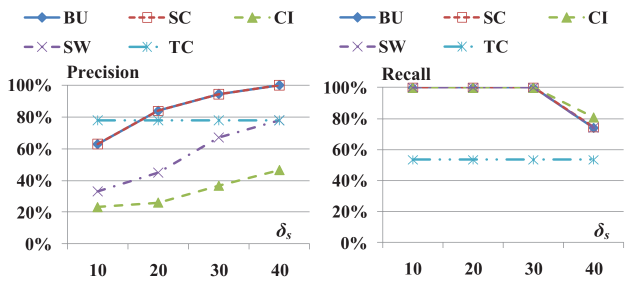


Fig. 20. Effectiveness: (a) precision, (b) recall vs. δ_s

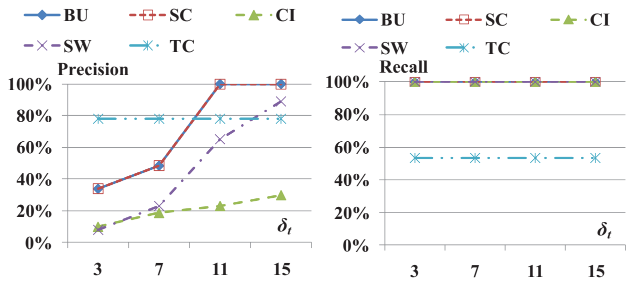


Fig. 21. Effectiveness: (a) precision, (b) recall vs. δ_t

VI. Discussions

In real world, the data items on streaming trajectories arrive in different timestamps with various delays. The objects may not report their positions at the same timestamp. With rigorous constraints on time, it is difficult to discover meaningful companions, the time cost is also high to process the companion discovery at every data item's arrival.

Suppose two travelers visit the same place in a very short time interval (*e.g.*, ten seconds), they can be seen as moving together. In this study, we utilize the concept of snapshot as the projection of all the objects' spatial information in a given time span. The snapshot can be formed in two ways:

- Equal length: A fixed time span is set according to the reporting frequency of the devices, and one snapshot is generated for each time span;
- Equal width: The minimum number of the objects in a snapshot is defined according to the concern of precision. The system does not generate new snapshot until enough objects have reported their positions.

The snapshots can be generated by the sliding window model in a batch processing mode. The sliding window stores the object positions, and the window size equals the object number. When a new data item arrives, the window updates the position of corresponding objects. Once the window satisfies the length or width requirements, the system clears the window, outputs all the data items to generate a snapshot and carries out the companion discovery task on that snapshot.

It is possible that there are multiple position reports for a single object in a snapshot. Fig. 22 shows an example of three snapshots. Object o_2 has multiple position reports in the time span of snapshot s_2. In such case, the sliding window updates o_2's position as the mean value of reported coordinates. Another problem is missing data. In Fig. 22, although o_3 may travel together with o_1 and o_2 in snapshot

s_1 and s_3, since its position record is missing in s_2, it will not be included in the companion. A robust system should tolerant such cases. In this study, we use the *inactive period* to deal with missing data. It is a threshold for the max interval between two position reports of the object. If the object is missing in a snapshot, as long as the inactive period is less than the threshold, the system still assumes that the object is traveling together with the companion in previous snapshot. In Fig. 22, if the inactive period threshold is larger than 16 seconds, the system assumes that o_3 travels together with o_1 and o_2 in s_2.

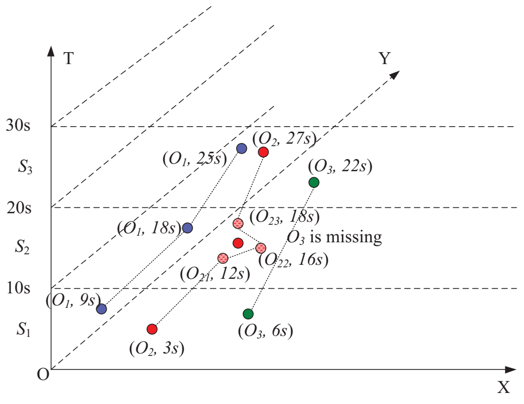


Fig. 22. Snapshots in the Trajectory Stream

We also conduct experiments to study the influence of the inactive period. Since SW and TC are not affected by this parameter, we run the algorithms of BU, SC and CI on dataset D_3 by tuning the inactive period threshold from 0 to 6 snapshots. Fig. 23 shows the algorithm's time and space cost. With larger inactive periods, all the algorithm's space costs increase since they cannot prune the candidates if several objects temporally leave the companion. The system has to spend more time to make intersections with a larger candidate set.

The effectiveness experiment is carried out on dataset D_2. Since D_2 is collected with high quality, there is no missing data for any object. We randomly remove 10% data from D_2, and use the remaining parts for experiments. The inactive period is changed from 0 to 6 snapshots, and other parameters are set as default values. As shown in Fig. 24 (a), the precision of companion discovery decreases with inactive period, since more false positive companions are generated. However, the

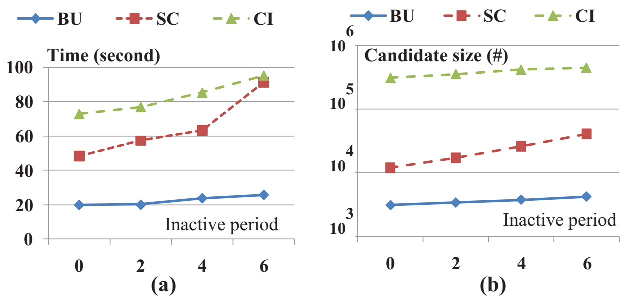


Fig. 23. Efficiency: (a) time, (b) space vs. inactive period

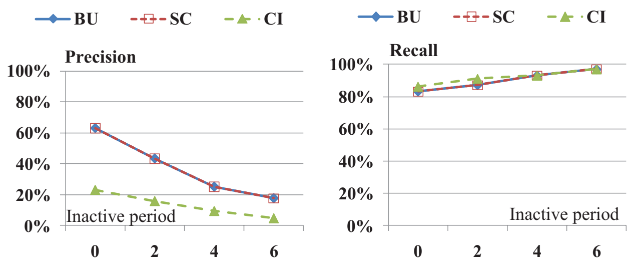


Fig. 24. Effectiveness: (a) precision, (b) recall vs. inactive period

recalls increase as the inactive period grows (Fig. 24 (b)). Even with 10% missing data, BU and SC can still achieve a high recall near 95%.

VII. Related Work

Gaffney *et al.* first proposed the fundamental principles of clustering moving objects based on the theories of probabilistic modelling [9], [6]. Lee *et al.* proposed a novel partition-and-group framework to find the clusters based on sub-trajectories [20]. Yang *et al.* proposed the idea of neighbor-based pattern detection method for windows [30]. Ester *et al.* made the progress to generate incremental clusters [7]. Zhang and Lin used the k-centre clustering algorithm [11] to discover interesting patterns [33]. More recently, Jensen *et al.* utilized the velocity features to cluster objects and improved the clustering effectiveness for trajectory data [16].

However, as pointed out in [17], the trajectory clustering methods cannot be used directly for traveling companion discovery. Most algorithms, such as [7], generate clusters from the whole dataset. If the clusters are not retrieved snapshot by snapshot, the companion patterns cannot be discovered from the object relationships among snapshots.

Movement pattern discovery is a hot topic in recent years. The problem has been variously referred to as the search for *flocks* [12], *moving clusters* [18], *spatial-tempo joins* [4], *spatial co-locations* [31], *meetings* [13], *convoys* [17], *moving groups* [3], *swarms* [24] and so on.

One of the earliest works is *flock* discovery [13]. A flock is defined as a group of objects moving together within a circular region [12]. There are several variations of this model: *Variable flock* permits the members to change during the time span [5], *meeting* is a circle similar to flock but fixed in a single location all the time [12]. However, such shapes are restricted to circles and the results are also sensitive to the parameter of radius.

Li *et al.* designed a flow scan algorithm for hot route mining [22]. Liu *et al.* mined frequent trajectory patterns by using RF tag arrays. Their work successfully demonstrated the feasibility and the effectiveness of movement patterns in real life [25]. Tao *et al.* proposed the technique of spatio-temporal aggregation using sketch index. This method can process the queries an order of magnitude faster than the previous works [29]. Giannotti *et al.* proposed the *interest region* based mining algorithm [10]. Zhang *et al.* propose the techniques to produce intersections of streaming moving objects [34]. This method is a big improvement from existing algorithms by the speed-up of several orders of magnitude. Nutanong *et al.* use a safe region to report objects that do not change over time [26]. The proposed V*-Diagram has much smaller IO and computation costs than previous methods. It outperforms the best existing technique by two orders of magnitude. However, since the above methods focus more on discovering hot spots, regions or routes rather than object groups, they cannot be used directly for companion discovery.

Kalnis *et al.* proposed the first study to automatic extract *moving clusters* from large spatial datasets [18]. In a recent work, Jeung *et al.* proposed the framework of *convoy* query [17]. It is a significant step forward in the works of movement pattern mining, since it allows the objects to organize in arbitrary shapes. Li *et al.* further released the constraints of convoy and proposed the *swarm pattern* to discover object groups in a sporadic way [24].

The concepts of convoy and swarm patterns are similar to traveling companion. The major differences are about the discovery algorithms. The convoy algorithm needs to scan the entire trajectory into memory to make trajectory simplification, and the system also needs to load the whole dataset into memory to search for swarms. Hence it is impractical to use them in a data stream environment. In addition, the swarm pattern is a frequent itemset-based concept. Since it is difficult to detect large size frequent itemsets [36], the swarm pattern has limited applicability for datasets with large scale objects.

VIII. Conclusion and Future Work

In this study we investigate the problem of traveling companion discovery on trajectory data streams. We propose the algorithms of smart-and-closed discovery to efficiently generate companions from trajectory data. The model of traveling buddy is proposed to help improve both the clustering and intersection processes for companion discovery. We evaluate the proposed algorithms in extensive experiments on both real and synthetic datasets. The buddy-based method is shown to be an order of magnitude faster than existing approaches. The effectiveness of buddy-based algorithm also outperforms other competitors in terms of precision and recall.

In the future, we plan to extend the companion discovery technique to more complex scenarios, such as road networks. We are also interested in integrating the methods to real application cases such as battlefield monitoring systems and traffic analysis services.

IX. Acknowledgements

The work was supported in part by U.S. NSF grants IIS-0905215, CNS-0931975, CCF-0905014, IIS-1017362, the U.S. Army Research Laboratory under Cooperative Agreement No. W911NF-09-2-0053 (NS-CTA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] http://foursquare.com/.

[2] http://www.movebank.org.

[3] H.-H. Aung. Discovering moving groups of tagged objects. In *Technique Report, National University of Singapore*, 2008.

[4] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *ACM GIS*, 2005.

[5] M. Benkert, J. Guddmundsson, F. Hubner, and T. Wolle. Reporting flock patterns. *Comput. Geom. Theory Appl.*, 41(3):111–125, 2008.

[6] I. V. Cadez, S. Gaffney, and P. Smyth. A general probabilistic framework for clustering individuals and objects. In *SIGKDD*, 2000.

[7] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, 1998.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, 1996.

[9] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *SIGKDD*, 1999.

[10] F. Giannotti, M. Nanni, D. Pedreschi, and F. Pinelli. Trajectory pattern mining. In *SIGKDD*, 2007.

[11] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, pages 293–306, 1985.

[12] J. Gudmundsson and M. v. Kreveld. Computing longest duration flocks in trajectory data. In *ACM GIS*, 2006.

[13] J. Gudmundsson, M. v. Kreveld, and B. speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In *ACM GIS*, 2004.

[14] J. Han and M. Kamber. *Data Mining: Concepts and Techniques Second Edition*. Morgan Kaufmann, 2006.

[15] S. Har-Peled. Clustering motion. *Discrete and Computational Geometry*, 31(4):545–565, 2003.

[16] C. S. Jensen, D. Lin, and B. C. Ooi. Continuous clustering of moving objects. *IEEE TKDE*, 19(9):1161–1174, 2007.

[17] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. In *VLDB*, 2008.

[18] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatial-temporal data. In *SSTD*, 2005.

[19] T. Krout. Cb manet scenario data distribution. In *Technique Report of BBN*, 2007.

[20] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, 2007.

[21] M. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, 2003.

[22] X. Li, J. Han, J.-G. Lee, and H. Gonzalez. Traffic density based discovery of hot routes in road networks. In *SSTD*, 2007.

[23] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *SIGKDD*, 2004.

[24] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters accurate discovery of valid convoys from moving object trajectories. In *VLDB*, 2010.

[25] Y. Liu, L. Chen, J. Pei, Q. Chen, and Y. Zhao. Mining frequent trajectory patterns for activity monitoring using radio frequency tag arrays. In *IEEE PerCom*, 2007.

[26] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: A query dependent approach to moving knn queries. In *VLDB*, 2008.

[27] L.-A. Tang, X. Yu, S. Kim, J. Han, C.-C. Hung, and W.-C. Peng. Tru-alarm: Trustworthiness analysis of sensor networks in cyber-physical systems. In *ICDM*, 2010.

[28] L.-A. Tang, Y. Zheng, X. Xie, J. Yuan, X. Yu, and J. Han. Retrieving k-nearest neighboring trajectories by a set of point locations. In *SSTD*, 2011.

[29] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, 2004.

[30] D. Yang, E. A. Rundensteiner, and M. O. Ward. Neighbor-based pattern detection for windows over streaming data. In *EDBT*, 2009.

[31] J. S. Yoo and S. Shekhar. A partial join approach for mining co-location patterns. In *ACM GIS*, 2004.

[32] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *GIS*, 2010.

[33] Q. Zhang and X. Lin. Clustering moving objects for spatial-temporal selectivity estimation. In *ADC*, 2004.

[34] R. Zhang, D. Lin, K. Ramamohanarao, and E. Bertino. Continuous intersection joins over moving objects. In *ICDE*, 2008.

[35] Y. Zheng, X. Xie, and W. Ma. *GeoLife: A Collaborative Social Networking Service among User, location and trajectory*. IEEE Data Engineering Bulletin, 2010.

[36] F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. In *ICDE*, 2007.